

https://dspace.org/

Design, Implementation and Performance Evaluation of an Elastic Partitioned Global Address Space Storage (EPGASS)

Ohene-Kwofie, Daniel

2025-10
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/50085
Ohene-Kwofie, Daniel. (2025). Design, Implementation and Performance Evaluation of an Elastic Partitioned Global Address Space Storage (EPGASS). [PhD thesis, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/50085

Downloaded from a DSpace repository. https://dspace.org/

UNIVERSITY OF THE WITWATERSRAND



Design, Implementation and Performance Evaluation of an Elastic Partitioned Global Address Space Storage (EPGASS)

Author:

Daniel OHENE-KWOFIE

Supervisors:

Prof. Scott HAZELHURST

Prof. Ekow OTOO

A thesis submitted to the Faculty of Engineering and the Built Environment in fulfilment of the requirements for the degree of

Doctor of Philosophy

Declaration

I, Daniel OHENE-KWOFIE, declare that this thesis titled, "Design, Implementation and Performance Evaluation of an Elastic Partitioned Global Address Space Storage (EPGASS)" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 02 October 2025

Abstract

Advances in modern technologies, high-speed and more precise instrumentation for data acquisition, cutting-edge high-performance computing, improved simulation modelling, and the development of large data centres have collectively contributed to the rapid accumulation of massive datasets, commonly referred to as Big Data. While the processing speed of CPU/GPGPU in high-performance computing has steadily increased, the rate of data input and output (I/O), even with parallel I/O, has lagged behind the advances in computing speeds. Although a number of computing techniques have been adopted to bridge the CPU I/O gap, I/O still remains a major bottleneck to high performance data processing and large scale scientific applications. Recent advances in database management and data processing technologies focus on in-memory (or DRAM) data storage system to enhance performance. The challenge, however, lies in developing an I/O subsystem infrastructure that can match the computational speeds of advanced computing systems operating at the petascale and exascale levels. The thesis proposes an I/O subsystem infrastructure leveraging in-memory computing to meet the computational speeds of advanced computing systems at the petascale and exascale level, leveraging the Partitioned Global Address Space Model (PGAS) of computing on commodity hardware. The PGAS system is used to host an in-memory data store, providing high performance access to in-memory data. The proposed architecture is reminiscent of that of RAMCloud, but with some major differences. The architecture consists of a number of commodity nodes, that can grow into hundreds or thousands, whose physical memories are aggregated as a single global logical address space, with a fraction of these nodes serving as the I/O nodes, while the remaining nodes form the collection of compute nodes. The I/O nodes maintain in-memory data in the global address space and run thousands of threads, some of which persist the memory resident data onto Solid-State Device (SSDs), attached to the nodes. The thesis refers to this architecture as *EPGASS: An Elastic Partitioned Global Address Space Storage* . The major contribution of the thesis is the design, implementation, and application usage of EPGASS. The study presents strategies to address challenges such as memory-to-memory data copying, fault-tolerance, long-term persistence with asynchronous parallel data migration from memory-to-SSDs and vice versa and also from SSDs-to-hard-disks (HDD), with a hierarchical storage approach. The thesis also discusses use-cases with diverse applications, including a novel approach to dense matrix-matrix multiplication and multidimensional index schemes, which minimise the communication overhead to improve algorithmic efficiency.

Acknowledgements

I extend my heartfelt gratitude to my supervisor, Prof. Scott Hazelhurst for his guidance, support and most of all, patience. I am truly grateful for the knowledge and insight, as well as the untold number of hours spent reviewing this work.

Special thanks to the MRC/Wits Agincourt team; Prof. Stephen Tollman, Prof. Kathy Khan, Prof. F. Xavier Gómez-Olivé, Dr. Chodziwadziwa Kabudula and many others in the School of Public Health for their immense support and continual motivation to finish this work. I am most grateful.

Am grateful to the Centre for High Performance Computing (CHPC) South Africa, grant number CBBI0930), and the National Energy Research Scientific Computing Center (NERSC), Berkeley, USA for providing the compute infrastructure and resources for the experiments conducted for this work.

To my lovely wife and family, thanks for your continual support and encouragement. I can honestly say that this would not have been possible if I had not received your support and encouragement throughout. Thank you.

In memoriam: I would like to acknowledge the invaluable contributions of the late Prof. Ekow Otoo who inspired and made invaluable contributions to this work. Our thoughts are with his family, friends, and colleagues during this time of reflection.

Contents

Declaration i

Abstract ii

Acknowledgements iii

List of Figures viii

List of Tables xii

Abbreviations xiii

List of Publications xvi

1 Introduction 1

1.1 Overview . . . 1

1.2 Goals of EPGASS . . . 5

1.3 Motivation . . . 5

1.4 Main Contributions . . . 7

1.5 Organisation Of Thesis . . . 8

2 Background And Related Work 9

2.1 Introduction . . . 9

2.2 High Performance Programming Models . . . 10

2.2.1 The Message Passing Model . . . 10

2.2.2 The Partitioned Global Address Space Model . . . 10

Unified Parallel C . . . 11

2.2.3 The Hybrid MPI+UPC Model . . . 13

2.3 I/O in Parallel Computing . . . 15
2.3.1 Parallel File Systems . . . 16
2.3.2 Parallel I/O Middleware . . . 18
MPI-I/O . . . 19
UPC - I/O . . . 20
2.3.3 The Adaptable I/O System (ADIOS) . . . 21
2.3.4 Parallel I/O Challenges . . . 22
2.4 Distributed In-Memory Systems . . . 23
2.4.1 Alluxio: A Virtual Distributed File System . . . 24
2.4.2 The RAMCloud . . . 25
2.4.3 Memcached . . . 26
2.4.4 DataSpaces . . . 27
2.5 Memory mapped file I/O . . . 28
2.6 In-memory Databases . . . 29
2.7 Summary . . . 29

3 The Elastic Partitioned Global Address Space System 31
3.1 Introduction . . . 31
3.2 Architectural Overview . . . 31
3.2.1 Input/Output Nodes . . . 32
3.2.2 Compute Node . . . 35
3.2.3 High-Speed Network Interconnects . . . 37
3.3 Hierarchical Storage Structure . . . 39
3.4 Fault Tolerance And Recovery . . . 40
3.5 Summary of EPGASS Architecture . . . 41

4 EPGASS I/O Performance Benchmark and Analysis 42
4.1 Introduction . . . 42
4.2 Benchmarking Tools . . . 43
4.2.1 The NAS Parallel Benchmarks . . . 44
4.2.2 The UPC STREAM micro-benchmark . . . 47
4.3 Experimental Results . . . 50
4.3.1 Experimental setup . . . 50

4.3.2 NAS Parallel Benchmarks Results . . . 50
4.3.3 STREAM micro-benchmark Results . . . 60
4.4 Chapter Summary . . . 62

5 Matrix-Matrix Multiplication **65**
5.1 Introduction . . . 65
5.2 Background . . . 67
5.3 Single Matrix-Block-Shift Algorithm . . . 71
5.3.1 Blocking SMBS Algorithm . . . 71
5.3.2 Non-Blocking SMBS Algorithm . . . 75
5.4 Analysis of the algorithm . . . 76
5.5 EPGASS and the Single Matrix-Block-Shift Algorithm . . . 79
5.5.1 The General UPC algorithm . . . 79
5.5.2 PGAS SMBS . . . 80
5.6 Performance Evaluation . . . 85
5.6.1 Experimental setup . . . 85
5.6.2 Result And Discussion . . . 86
5.6.3 EPGASS SMBS . . . 93
5.7 Chapter Summary . . . 99

6 *k*-d Trees And EPGASS **101**
6.1 Introduction . . . 101
6.2 Background . . . 102
6.3 Application of *k*-d trees . . . 104
6.4 The Spatial Median *k*-d tree structure . . . 105
6.5 Parallel *k*-d tree Algorithm . . . 107
6.5.1 Tree Construction . . . 107
6.5.2 Tree Construction Optimisations . . . 107
6.6 *k*-d tree Search Algorithms . . . 108
6.6.1 Orthogonal Range Search Algorithm . . . 108
6.6.2 Radius Range Search Algorithm . . . 109
6.6.3 Nearest Neighbour Algorithm . . . 110
6.7 Performance Evaluation . . . 114

6.7.1 Experimental setup 114
6.7.2 Results And Discussion 115
Building the *k*-d tree Index Structure 115
Search Operations on the *k*-d tree Index Structure 120
6.8 Chapter Summary 124

7 Conclusion And Future Direction **126**
7.1 Summary of Research contributions 126
7.1.1 The Elastic PGAS concept And Performance Benchmarks 126
7.1.2 The Single Matrix Block Shift (SMBS) algorithm And EPGASS . . 128
7.1.3 *k*-d Trees And EPGASS 130
7.2 Future Directions 131

A Appendix **133**

References **152**

List of Figures

1.1 Global Datasphere 2

2.1 High-level system diagram of the UPC Compiler – designed for Portability 12

2.2 UPC memory layout highlighting partitioned storage – Shared Global address space and Private spaces. 13

2.3 MPI+UPC hybrid execution models. smallThe grey circles denote hybrid MPI+UPC processes, while the white circles represent UPC-only processes. 15

2.4 Internal System Architecture Of BeeGFS 17

2.5 Parallel I/O Stack 18

2.6 ADIOS 2 21

2.7 Fundamental approaches for handling I/O in parallel applications 23

2.8 Diagram of the distributed shared memory abstraction, depicting the mapping of memory across multiple compute nodes into a single global address space accessible by all processes. 24

2.9 RAMCloud Architecture 25

2.10 Sample Memcached setup 27

2.11 DataSpaces architecture showing the structured layers and components – the distributed in-memory associative object store, scalable messaging, as well as the runtime mapping and scheduling. 28

3.1 Schematic Diagram of the Architecture of EPGASS highlighting the compute and I/O nodes setup with commodity hardware which leverages PGAS model for in-memory application processing and data storage 37

3.2 Software system stack of EPGASS highlighting the various systems leveraged for applications and data processing 38

3.3 Hierarchical Storage Structure 40

4.1 Layout of testing cluster codenamed "Jaguar" 44
4.2 Evaluation of system performance using the **EP** benchmark comparing UPC and MPI implementations 53
4.3 Evaluation of system performance using the **IS** benchmark comparing UPC and MPI implementations 54
4.4 Evaluation of system performance using the **FT** benchmark comparing UPC and MPI implementations 55
4.5 Evaluation of system performance using the **MG** benchmark comparing UPC and MPI implementations 56
4.6 Evaluation of system performance using the **CG** benchmark comparing UPC and MPI implementations 57
4.7 Proportion of total execution time attributed to communication in the **IS** benchmark showing increasing % of time for communication with increase in number of processors. 58
4.8 Memory utilisation. (A) shows the relative memory usage of the UPC benchmark applications compared to MPI, whiles (B) illustrates the actual usage in gigabytes 59

5.1 Initial Data Allocation for Cannon's Algorithm on 3×3 Grid 69
5.2 Block of Matrices A and B 73
5.3 Block Distribution of Matrices A and B 73
5.4 Sketch of SMBS Algorithm 75
5.5 Computational time and Operational throughput for various Matrix Sizes using 2500 cores 87
5.6 Computational time and Operational throughput for various Matrix Sizes using 900 cores 88
5.7 Computational time and Operational throughput for various Matrix Sizes using 400 cores 89

5.8 Evaluation of algorithms' scaling efficiency. This experiment kept a fixed problem size of 10800 × 10800 and increased the number of processors from 100 to 2500 cores in the strong scale test. In the weak scale test, the workload was increased from 60 × 60 to 14400 × 14400 matrices, while the processors increased from 25 to 6400 cores 91

5.9 Scaling efficiency using 4000 × 4000 Matrix with Variants of SRUMMA 92

5.10 Average time and throughput for various Matrix workloads for 256 cores 94

5.11 Average time and throughput for various Matrix workloads for 196 cores 95

5.12 Evaluation of algorithms' scaling efficiency. The strong scale test maintains a fixed problem size of 10800 × 10800, while the number of processors is increased to 256 cores. For the weak scale test, however, the workload is increased from 60 × 60 to 14400 × 14400 matrices, while the number of processors are doubled from 16 cores up to 256 cores 96

5.13 Evaluation of algorithms' scaling efficiency with CFAIR-100 image processing dataset 98

6.1 Increasing timing for Initial data distribution across processors 116

6.2 *k*-d tree Index Build time for varying workloads and cores 117

6.3 *k*-d tree Index Build time with 1,048,576 GIS dataset for varying and cores . . 118

6.4 Evaluation of *k*-d tree construction scaling efficiency. For strong scaling, the workload size was fixed at 1048576. For weak scaling, the workload was increased from 4096 to 1048576, while the number of processors was doubled from 16 to 256. *Note: lower is better. Discrete points are connected to illustrate performance trends as processor count increases.* 119

6.5 *k*-d tree Index Build Speed-up 120

6.6 *k*-d tree Index Build Scaling efficiency 121

6.7 *k*-NN Search time for varying workloads and cores 122

6.8 *k*-NN Search time using 1,048,576 dataset for varying and cores 122

6.9 Evaluation of *k*-NN scaling efficiency between FLANN & KDT in $\log_{10}$ scale. In the strong scaling test, a fixed size workload of 1048576 was maintained. In the weak scaling test, the workload was doubled from 4096 to 1048576, while the number of processors was simultaneously doubled from 16 to 256. *Note: lower is better* 123

A.1 An illustrative image showing the concept of convolution where, a 3×3 matrix kernel is applied to the input image to produce the convolved feature which is then forwarded to the next layer 133

A.2 Typical layout of the convolutional neural network highlighting feature extraction layer where kernel is applied 133

List of Tables

4.1 The NAS Parallel Benchmarks Workload Classes 48

4.2 UPC STREAM Benchmark highlighting memory bandwidth for Strided access for Reads & Writes 61

4.3 UPC STREAM Benchmark highlighting memory bandwidth for Random Reads & Writes 62

6.1 Sample queries in space: where P is a set of data points in $\mathbb{R}^k$ space, and $\mathbf{q}$ a search/query point in $\mathbb{R}^k$. 102

List of Abbreviations

ADIOS	**A**daptable **I**nput **O**utput **S**ystem
AI	**A**rtificial **I**ntelligence
ANSI	**A**merican **N**ational **S**tandards **I**nstitute
BDAS	**B**erkeley **D**ata **A**nalytics **S**tack
BLCR	**B**erkeley **L**abs **C**heckpoint **R**estart
CHPC	**C**entre for **H**igh **P**erformance **C**omputing
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CSV	**C**omma-**S**eparated **V**alues
DIMM	**D**ual **I**n-line **M**emory **M**odule
EOD	**E**xperimental and **O**bservational **D**ata
FTI	**F**ault **T**olerant **I**nterface
FLOPS	**Fl**oating-Point **O**perations **P**er **S**econd
GbE	**G**iga**b**it **E**thernet
GIS	**G**eographic **I**nformation **S**ystem
GPFS	**G**eneral **P**arallel **F**ile **S**ystem
GPGPU	**G**eneral-**P**urpose **G**raphics **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
HDD	**H**ard **D**isk **D**rive
HDFS	**H**adoop **D**istributed **F**ile **S**ystem

HPC **H**igh **P**erformance **C**omputing

IDC **I**nternational **D**ata **C**orporation

I/O **I**nput **O**utput

IoT **I**nternet **o**f **T**hings

iWARP **I**nternet **W**ide-**A**rea **R**DMA **P**rotocol

***k*-NN** **k**- **N**earest **N**eighbors

LRU **L**east **R**ecently **U**sed

NERSC **N**ational **E**nergy **R**esearch **S**cientific **C**omputing **C**enter

MFLOPS **M**ega **F**loating-point **O**perations per second

MMIO **M**emory **M**apped **I**nput **O**utput

MPI **M**essage **P**assing **I**nterface

NAS **N**ASA **A**dvanced **S**upercomputing

NPB **N**AS **P**arallel **B**enchmark

NVM **N**on-**V**olatile **M**emory

NVRAM **N**on-**v**olatile **R**andom **A**ccess **M**emory

OSI **O**pen **S**ystems **I**nterconnection

OSM **O**pen **S**treet **M**ap

PFLOPS **P**eta **Fl**oating-Point **O**perations **P**er **S**econd

PGAS **P**artitioned **G**lobal **A**ddress **S**pace

PGEMM **P**arallel **GE**neral **M**atrix **M**ultiplication

PUMMA **P**arallel **U**niversal **M**atrix **M**ultiplication

RAM **R**andom **A**ccess **M**emory

RDMA **R**emote **D**irect **M**emory **A**ccess

RMA **R**emote **M**emory **A**ccess

RPC **R**emote **P**rocedure **C**all

SATA **S**erial **A**dvanced **T**echnology **A**ttachment

SCR **S**calable **C**heckpoint/**R**estart

SKA **S**quare **K**ilometre **A**rray

SKAO **S**quare **K**ilometre **A**rray **O**bservatory

SPMD **S**ingle **P**rogram, **M**ultiple **D**ata

SRUMMA **S**hared and **R**emote memory access based **U**niversal **M**atrix **M**ultiplication **A**lgorithm

SSD **S**olid-**S**tate **D**rive

SSIO **S**torage **S**ystem **I**nput **O**utput

SUMMA **S**calable **U**niversal **M**atrix **M**ultiplication **A**lgorithm

TFLOPS **T**era **F**loating-point **O**perations per second

UPC **U**nified **P**arallel **C**

XML e**X**tensible **M**arkup **L**anguage

List of Publications

1. Ohene-Kwofie, D., Hazelhurst, S. (2024). **Single Matrix Block Shift (SMBS) Dense Matrix Multiplication Algorithm**. In: Gerber, A. (eds) South African Computer Science and Information Systems Research Trends. SAICSIT 2024. Communications in Computer and Information Science, vol 2159. Springer, Cham. https://doi.org/10.1007/978-3-031-64881-6_11

2. Ohene-Kwofie, D., Otoo E. (2015). **PGAS in-memory data processing for the Processing Unit of the Upgraded Electronics of the Tile Calorimeter of the ATLAS Detector**. Journal of Physics: Conference Series, vol 645, https://dx.doi.org/10.1088/1742-6596/645/1/012022

Chapter 1

Introduction

1.1 Overview

The growing disparity between CPU and I/O speeds poses a significant challenge to overall performance in High Performance Computing (HPC), particularly as petascale and exascale computing increasingly become standard in scientific computing. Although the computing power of multicore systems has been growing, the performance of storage subsystems has lagged behind the increasing performance demands of applications resulting from parallel computing and architectural improvements [1].

HPC applications enable simulating scientific phenomena at finer granularities and massive scales by taking advantage of advances in parallel processing architectures. Finer granularities mean larger amounts of data, resulting in a higher data growth rate in size and complexity, requiring efficient strategies to manage the data on storage file systems. This has led to a noticeable performance gap between processing speeds and the capabilities of the storage I/O subsystem. Storage devices are increasing in capacity, but not necessarily in performance [2]. This performance gap is further exacerbated by the rapid growth in data volumes resulting from solutions to today's complex scientific and industrial problems.

Poorly performing I/O subsystems often constrain the scalability of applications. Fast and efficient parallel I/O is essential for the success of several HPC applications, including emerging fields like deep learning, which has surged in importance in recent

times for addressing crucial challenges and forecasting trends across various disciplines. These include natural language processing, computer vision, speech recognition, and climate science [3–7].

With advancements in Artificial Intelligence (AI), mobile devices, autonomous driving, and the Internet of Things (IoT), the volume of data being generated, collected, stored, managed, and analysed globally is increasing exponentially. The International Data Corporation (IDC) has estimated the Global Datasphere to reach 175 Zettabytes(ZB) [1] in 2025 [8]. Statista has projected this to grow to 394 ZB by 2028 [9]. Such explosive data growth requires efficient storage infrastructure and fast processing to gain insight and make quality business decisions. Figure 1.1 shows the growing global data-sphere.

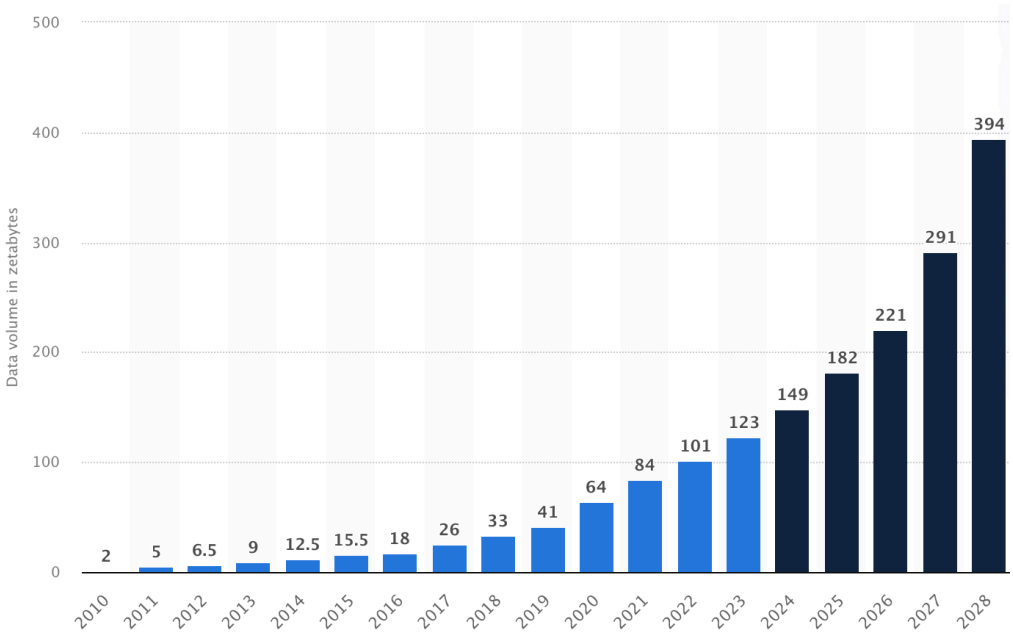


FIGURE 1.1: Increasing global data volume estimated to reach over 390 ZB by 2028 [9]

Processing such huge amounts of data presents unique challenges, especially with the I/O subsystem, even in high-performance computing clusters. While several data-intensive applications are processed on HPC clusters, resource contention on the shared storage systems connected to the clusters becomes a major issue. This contention leads

[1] 1 ZB = 1000^7 bytes = 10^{21} bytes = 1000 exabytes = 1 million petabytes = 1 billion terabytes = 1 trillion gigabytes

to I/O performance degradation and diminishes the performance gains achieved by coordinated parallel I/O subsystems, including optimizations offered by MPI-I/O implementations [10]. The performance of many of these systems is, therefore, constrained by the I/O subsystem [11]. While caching can greatly enhance read performance, it offers limited benefits for write performance. This is largely because highly parallel systems need to ensure fault-tolerance, which is mostly achieved by replicating the data written across the nodes [12]. This synchronous replication process also introduces some computational overhead resulting in overall performance degradation. However, a complete distributed in-memory data store could ensure a significant read/write performance improvement, especially with asynchronous background writes to persistent storage.

In recent times, solid-state devices (SSDs) or flash storage systems have gradually been embedded in disk-based storage as a solution to addressing the poor performance regarding the performance of disk storage. It is now common place to see over $12 \sim 24$ TB disks incorporated with 512GB or over 1 TB SSDs. However, a shift of primary data storage to main memory, from disk, where disk is relegated to a persistent backup/archival role provides an extremely fast performance [13–16]. In-memory data processing systems deliver exceptionally fast response times and high throughput.

Ousterhout et al. [13] estimates that a complete Random Access Memory (RAM) storage system provides $\sim 100 \times -1000\times$ lower latency, and subsequently up to $\sim 1000\times$ greater throughput compared to a disk-based storage system. Since memory is volatile, effective replication and backup techniques are essential to ensure durability and availability of transactions comparable to those of disk-based systems. Fast data processing is of great importance in today's data centres and scientific computations, especially with big data application stacks such as Berkeley Data Analytics Stack (BDAS)[17], Apache Spark[18], and the Hadoop Ecosystems[19]. Efficient use of high-performance computing has the potential to significantly decrease the duration required for DNA sequencing, for example, and consequently make crucial data available for chemical analysis, potentially enabling doctors to accelerate the diagnostic process. However, I/O bottlenecks must be minimised from compute clusters to achieve this vision.

Despite being easy to implement, the general-purpose network storage available today cannot keep up with the throughput demanded by the compute nodes. When storage continues to be a bottleneck, it is not always beneficial to add more compute power. Though several high-end HPC storage solutions could in theory address the performance bottleneck, but are either too costly or too difficult to implement and manage. Without a viable mid-range HPC storage solution, the ability to cost-effectively proliferate rapid diagnosis and improve health outcomes would thus be limited. As noted by the Advanced Scientific Computing Advisory Committee's (ASCAC) report on exascale computing [20], numerous other market changing innovations will remain concepts, without affordable, high performance HPC storage capable of satisfying the data intensive needs of scientific, as well as commercial breakthrough simulation and modelling applications.

The thesis therefore proposes the implementation of an I/O subsystem and processing technique to mitigate the gap between computational speed and data access speeds in high performance computations and applications using commodity hardware. The study present the design and implementation of an in-memory data store which is based on the Partitioned Global Address Space (PGAS) programming model of computation that makes accessible a large logical global storage formed from logically aggregating the physical main memories of a collection of nodes. The projection is that this mechanism will reduce the I/O bottleneck of data-intensive HPC applications while providing considerable global address space to store massively large datasets from the aggregated physical memories of hundreds to thousands of I/O nodes. This is referred to as *EPGASS: An Elastic Partitioned Global Address Space Storage*. The targeted applications include scientific computations which have very high memory footprint as well as large data storage needs which goes beyond the memory capacities of individual nodes – which is very typical in many recent scientific applications and simulations, including genomic sequencing, image processing systems, and many more.

The thesis proposes a cost-effective in-memory storage system using commodity systems for applications that utilises the widely used HPC models such as the MPI and UPC for computation. The approach of this study leverages the strength of the hybrid MPI-UPC programming model. The architecture seeks to enhance the use of main

memory for dynamic memory intensive applications and provide a mechanism for UPC-based applications to dynamically allocate memory across compute nodes which join the cluster to contribute their memory resources. The architecture seeks to provide an efficient in-memory data store for fast processing while ensuring durability by using SSDs to provide high speed *intermediate* storage. The slow disk-based systems are used only for archival or replication purposes, similar to that of the RAMCloud [13]. This elastic aggregated memory of *EPGASS* is thus expected to enable pragmatic commercial, academic, and life sciences applications to meet their demands of very fast computations, higher job iteration rates and more precise insights with improved resolution accuracy.

1.2 Goals of EPGASS

The design and implementation of *EPGASS* is guided by the following goals:

- Ensure seamless dynamic memory allocation for applications across the nodes of an HPC cluster
- Provide high-performance asynchronous fault-tolerant and resilient I/O to allow for the implementation of scalable applications.
- Provision Hierarchical Data Storage paradigm including main-memory, SSDs, and persistent storage
- Provide network latency hiding technique to ensure high performance network communication between I/O nodes

1.3 Motivation

As high performance computing becomes significant in data-intensive research, it is critical to develop fast storage systems that complement the computational speed of the hardware. Mainstream database systems have had the need for speed in recent times. New development of database technologies focus on in-memory databases. Examples of such systems are Oracle's Timesten [21], IBM's solidDB [22] and VoltDB [23] and

SAPHANA [24]. However, these are reliant on SQL queries, which are not typical of the demands of high-performance computing applications. An assessment of the Square-Kilometre Array(SKA) project [25, 26] data management requirements shows that the current data management technologies may not cope with the data rates anticipated for processing and storing of the data. For instance, an average of 8 terabits per second of data will be transferred from the SKA telescopes, while the SKA observatory(SKAO) will archive 710 petabytes of data per year [26]. The upgraded Tile Calorimeter of the ATLAS detector operated at the Large Hadron Collider (LHC) at CERN is estimated to sustain $\sim$ 34TB/s of digital throughput [27]. These and other large data acquisitions and storage requirements in other sciences have motivated the need for new research in data management technologies. Many such complex scientific applications and large-scale computing systems demand substantial persistent storage alongside extensive in-memory data processing capabilities at exceptionally high speeds. These needs can only be met with large-scale parallel processing and in-memory resident data storage.

The key research problem definition is whether an I/O subsystem can be configured to match the computational speed and how to configure the system for a large number of I/O nodes; how to address fail-over and fault tolerance such that failed nodes do not halt a job which has been running for days or even weeks or months, as well as attaching more than one parallel hybrid application to the same set of I/O nodes.

The goals and objectives of this thesis are in the design, implementation, performance analysis and application usage of an elastic in-memory data storage that leverages the PGAS computational model based on MPI [28, 29] and the Berkeley-UPC [30]. The project seeks to develop a scalable high-performance data management technique that provides support for high-performance accesses to large datasets efficiently to enhance application performance in limiting I/O bottleneck. The main idea is to configure hundreds and thousands of nodes into a cluster that combines a substantial portion of the physical memories of the compute nodes into a logical global memory for in-memory data storage and processing. Similar ideas have been explored in the RAM-Cloud [31, 13] concept. However, the approach of this study is different in the sense that:

- It leverages MPI-UPC as the main model of computing and use libraries such as GASNet, Data-spaces [32, 33] for one-sided communication (directly accessing memory locations in another process without the need for explicit coordination or synchronization) and memory-to-memory data-copy respectively.

- The study focuses initially on applications that utilise hybrid programming with MPI and UPC. This will later be extended to accommodate applications that use MPI-OpenMP and MPI-OpenShmem hybrid programming.

- The target applications are those that conduct big-data analytics and processing, as well as process streaming data.

Moreover, whereas RAMCloud focuses on web-based applications as well as cloud storage systems [13], EPGASS focuses on high performance computations with the aim of enhancing memory usage for faster performance and application processing, thus enabling various complex data manipulations, particularly in scientific applications. Unlike RAMCloud that has specialised design protocols for interconnects hardware, the study leverage on the capabilities of Remote Data Memory Access (RDMA) capable interconnect such as Infiniband or 10 GbE switch running RDMA over Converged Ethernet (RoCE) [34] or Internet Wide Area RDMA Protocol (iWARP) [35].

1.4 Main Contributions

The major contribution of this thesis is the design and implementation of elastic in-memory data storage that leverages the PGAS model of computing. The main results being reported include the following.

- The implementation of an Elastic PGAS

- I/O benchmarks of EPGASS architecture and storage approach

- An analytical and experimental performance analyses of EPGASS approach.

- The implementation and application usage in large index schemes such as *k*-d Trees, LSD-Tree with EPGASS

- The application of EPGASS to dense matrix-matrix multiplication. The thesis also presents a novel approach to block dense matrix-matrix multiplication, which limits the number of data movements and, thereby reducing internode network latencies. This is referred to as the single-matrix block shift matrix multiplication (SMBS) algorithm

1.5 Organisation Of Thesis

The rest of the Thesis is organised as follows: Chapter 2 presents the background and related work on parallel I/O as well as data management and storage techniques employed in today's scientific applications. The chapter further discusses the different programming models used in many scientific applications. Chapter 3 presents the details of the EPGASS architecture as it pertains to the methodology and discusses the various components as well as the software leveraged in the design and implementation. Chapter 4 discusses performance evaluation techniques and I/O Benchmarks used for the evaluation of the system. In Chapter 5, the study presents the application and performance evaluation of a block dense matrix-matrix multiplication based on the EPGASS. Chapter 6 explores the application of EPGASS with large multidimensional index-based structures, such as the *k*-d Trees. Chapter 7 provides a summary of this thesis, along with a description and direction of future work.

Chapter 2

Background And Related Work

2.1 Introduction

As new challenges emerge in HPC and scientific computations, scientists have sought better and efficient ways of addressing them, particularly where it involves very large datasets. The hybrid approach to solving problems in HPC has been explored extensively in the literature [36–40] . It is widely believed among the HPC community that this approach is one of the sure paths to Exascale computing [41, 40]. The hybrid architecture of HPC systems, which could be characterized by distributed memory across nodes and shared memory with non-uniform memory access within each node, drives the development of hybrid approaches to problem-solving. The hybrid MPI+UPC approach discussed here (Section 2.2.3) exemplifies how combining paradigms can address communication overheads and scalability challenges, while aligning with the hardware realities of modern clusters. Within the EPGASS environment, applications stand to benefit from the advantages offered by this hybrid programming approaches. One key to reduced communication is with the use of RDMA that decouples inter-node data communication from the CPU. As such, an RDMA capable network is essential to the EPGASS configuration. Meanwhile, I/O bandwidth still poses challenges to fast computation. Section 2.2 examines the HPC programming models, after which I/O and data management concepts are discussed.

2.2 High Performance Programming Models

2.2.1 The Message Passing Model

MPI is one of the widely used and highly optimized parallel programming models used in the majority of high-performance computing systems [41]. It has been leveraged to solve numerous scientific problems, including molecular dynamics, 2D finite difference, atmosphere-modelling, as well as several simulations in many disciplines. The MPI standard supports two-sided messaging, enabling pairs of processes to exchange messages using *Send* and *Recv*, along with a range of powerful and efficient collective operations. Recent versions of the MPI standard support for one-sided messaging, which allows processes to remotely access designated memory segments of other nodes. As a result, processes collectively expose memory windows for remote access, which can be accessed by other processes in either active or passive target modes. In active mode, the host explicitly synchronizes its memory window during accesses, whereas in passive target mode, remote processes can access the window with no direct dialogue from the host.

The current MPI passive mode offers a model akin to UPC's global address space programming model. However, although it allows one-sided access to remote memory, it imposes stricter limitations compared to a global address space regarding consistency, coherence, and synchronization properties [42, 43]. The restrictions, however, have allowed MPI-2+ to be widely implemented on a range of systems, which include those lacking cache-coherent memory architectures. However, the portability constraints hinder programmers from fully leveraging the productivity and performance benefits offered by the hardware, particularly on systems with memory coherence. Further, several data-intensive computations have memory (storage, for that matter) requirements which go beyond that of single compute nodes, and can leverage on distributed shared-memory approach.

2.2.2 The Partitioned Global Address Space Model

The partitioned global address space (PGAS) is a parallel programming model that assumes a global memory address space, which is logically partitioned and distributed

among processors, such that portions are local to each process or thread [44]. Segments of the shared memory space may have affinity to a particular process or thread, which allows the process to leverage the locality of reference. Examples of PGAS model languages and libraries are the Unified Parallel C (UPC) [30], UPC++ [45], Fortress [46], Chapel [47], X10 [48], and Coarray Fortran [49], as well as Global Arrays [50]. The thesis discusses details of UPC, which is explored in this research, in the next section. PGAS aims to merge the benefits of the Single Program, Multiple Data (SPMD) programming model used in distributed memory architectures (such as in MPI) with the data-referencing paradigm of shared-memory systems. Unlike a traditional shared memory model with a single flat address space, PGAS provides a more practical approach by allowing the address space partitioning to reflect hardware-specific data locality. This study leverages the UPC Runtime to provide dynamic distributed global shared memory for application processing.

Unified Parallel C

UPC is an explicit parallel extension of ANSI C designed for high performance computing on large-scale parallel machines for distributed shared memory parallel programming [30]. It is based on the PGAS model, combining the programming simplicity of the shared memory paradigm with the scalability of the distributed memory paradigm. UPC offers an abstraction of a global address space that is evenly distributed across threads. Each shared data element in UPC is associated with a specific processor, and the model provides this data locality information to programmers, enabling them to optimize performance. Shared data can be accessed using built-in language-level features or through explicit one-sided communication operations. Users can access any data in the global address space, regardless of its location, without requiring explicit assistance from the owning thread/processor.

UPC utilizes the Global-Address Space Networking (GASNet), a high-performance communication software interface that is both network- and language-independent for high performance communication. Serving as the primary communication layer for UPC, GASNet delivers high performance, interface portability, and expressiveness for

the UPC language. The GASNet interface is divided into the *GASNet core API* as well as the *GASNet extended API*. The core API is based on the Active Message paradigm [51].

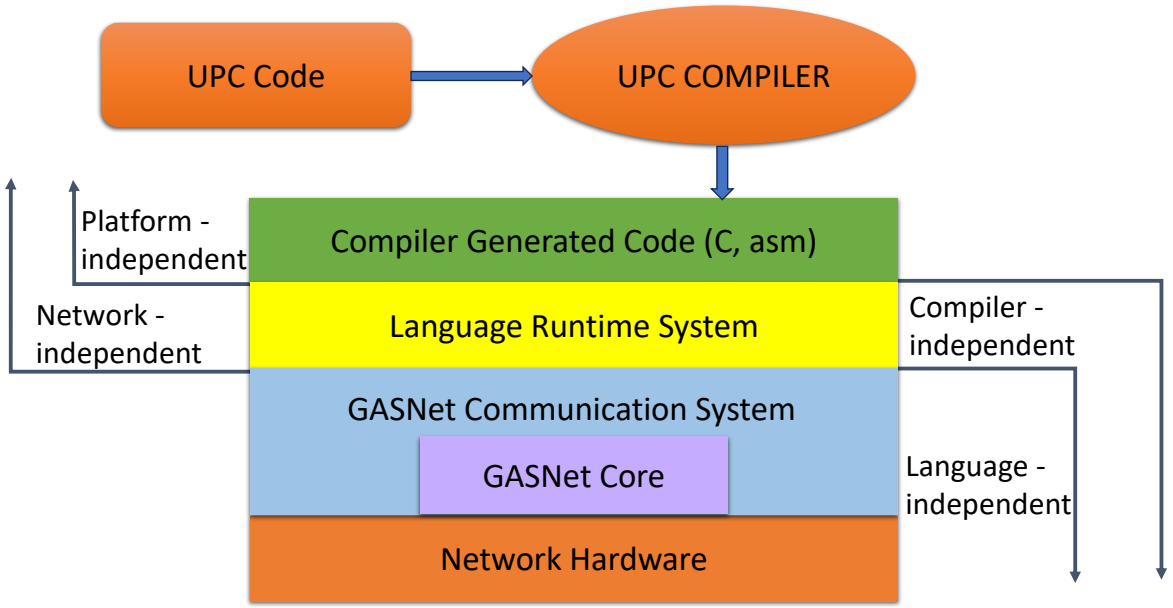


FIGURE 2.1: High-level system diagram of the UPC Compiler – designed for Portability [52]

As an extension of ANSI C, UPC provides a tunable approach to optimize performance. A programmer can begin with a sequential program, convert it into a basic shared-memory implementation, and then incrementally improve performance by optimizing data locality, adjusting array distributions, and relaxing the memory model [41]. The programmer can further optimize performance-critical sections of the code by employing explicit memory management, one-sided communication, and fine-tuning data distribution and locality control. Unlike MPI, UPC's model assumes memory coherence and supports fine-grained, asynchronous communication and dynamic distributed data structures, which pose challenges in the MPI-2 model. The MPI model is typically regarded as offering one-sided messaging rather than implementing a global address space [41], which is very much needed in memory constrained applications. Figure 2.1 highlights the basic layout of the UPC Compiler.

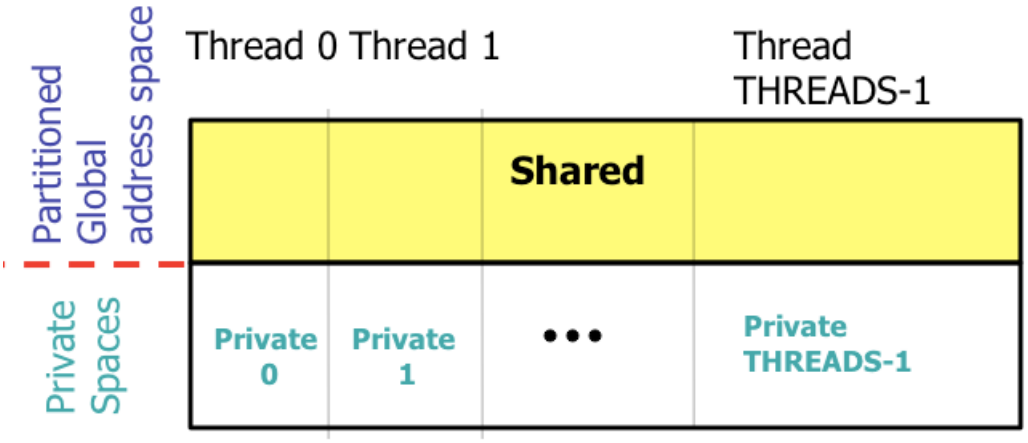


FIGURE 2.2: UPC memory layout highlighting partitioned storage – Shared Global address space and Private spaces.

UPC's memory layout consists of the private address space which is for an individual thread's private data and also the shared space. Figure 2.2 depicts the memory layout of UPC. Memory space is divided into partitions, with each partition P_i associated with thread i. The "shared" qualifier is used to access the global memory. Private variables are accessible by only one thread and offer performance benefits compared to shared variables [52], since they are readily accessible by the thread locally without remote access. UPC simplifies data distribution through the use of a distributed shared memory model. For example, to evenly distribute an array of size N in UPC, the user declares the array as "shared", and UPC automatically distributes it among threads in a round-robin manner. However, UPC lacks support for process groups and its distributions do not offer the flexibility required to allocate distributed shared arrays across a subset of processors. The hybrid MPI-UPC model, which offers the best of both MPI and UPC is discussed in Section 2.2.3.

2.2.3 The Hybrid MPI+UPC Model

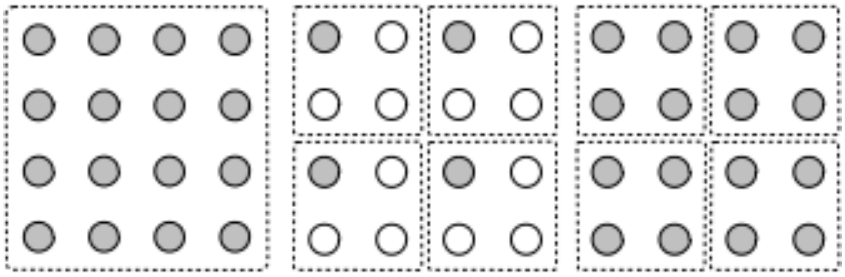
As new challenges emerge in HPC and scientific computations, scientists continually seek for better and efficient ways of addressing them. The hybrid approach to problem-solving in HPC has been adopted by scientists as an effective model for problem-solving because the model combines the best of several individual models to provide a unified approach to a solution.

The hybrid MPI+UPC programming model is one of such hybrid approaches to parallel computation that couples MPI and UPC in a program. This grants MPI programs access to a large, distributed shared global address space, offering notable advantages over MPI-2's one-sided communication. Additionally, UPC programmers can leverage an extra level of locality control to create multiple UPC spaces interconnected through MPI. As UPC is an extension of the C language and MPI is a library, the model can be described as a UPC program that invokes MPI library routines. The hybrid program is compiled using a UPC compiler and linked with MPI libraries. MPI requires large granularity for efficiency, as small messages incur high costs due to the fixed startup overhead latency associated with each communication. Thus, in this hybrid model, MPI is utilized for outer parallelism, while UPC is utilized for inner parallelism.

Dinan et al. categorized the MPI+UPC hybrid model into three submodels: *Flat*, *Nested Funnel*, and *Nested Multiple*. These classifications are based on the degree of nesting and the number of instances. The Flat model represents a non-nested setup with shared execution of MPI and UPC. In this model, each process participates in both MPI and UPC, possessing a unique MPI rank and a UPC thread identifier. The *Nested Funnel* and *Nested Multiple* models are nested configurations in which multiple UPC threads are initiated within a single MPI environment [41]. The communication within UPC processes or groups is managed exclusively through UPC, while intergroup communication is facilitated by MPI. Figure 2.3 illustrates the schematic layout of the various models.

A hybrid MPI+UPC program can be parameterized using both UPC thread identifiers and MPI ranks. The model utilizes the UPC constants *THREADS* and *MYTHREAD*, which specify the number of UPC threads during UPC program runs. Furthermore, each process in the outer MPI execution has a rank within the MPI *WORLD* communicator. As multiple UPC groups are launched in the *nested* model, the execution is further parametrized by a group rank and the number of groups. The hybrid execution can thus be fully described by the group, UPC THREAD ID, and MPI rank [41],

$$ID = \langle id_{group}, id_{UPC}, id_{MPI} \rangle$$



(A) Flat (B) Nested-Funneled (C) Nested-Multiple

FIGURE 2.3: MPI+UPC hybrid execution models.
smallThe grey circles denote hybrid MPI+UPC processes, while the white circles represent UPC-only processes.

and by the number of UPC groups, UPC group size and MPI communicator size,

$$N = \langle n_{group}, n_{UPC}, n_{MPI} \rangle$$

EPGASS leverages on this hybrid models to provide a fast, efficient storage mechanism for scientific computations and in-memory storage.

2.3 I/O in Parallel Computing

Parallel I/O can be implemented in a programming paradigm as operating system-based or library-based. The OS-based interfaces are advantageous since they have direct access to low-level operating system services. Nonetheless, they face portability challenges, as operating systems differ between vendors.

Parallel I/O subsystems generally comprise multiple layers, including middleware libraries and hardware components. In modern HPC systems, the typical parallel I/O stack includes high-level I/O libraries and file formats (e.g., HDF5, NetCDF, ADIOS), I/O middleware (e.g., MPI-I/O, POSIX), parallel file systems (e.g., Lustre, GPFS, OrangeFS), as well as storage and I/O hardware [53]. Scientific applications typically utilize I/O to accomplish tasks such as storing numerical simulation outputs for subsequent analysis, loading initial conditions or datasets for processing, check-pointing to save application states in case of system failures, and employing 'out-of-core' techniques for handling data volumes that exceed system memory capacity. The challenge with I/O is that scientific problems are increasing becoming data-intensive, involving

"big" data which require fast and efficient access. Some simulation applications can require several terabytes of data, and possibly generate several petabytes of data.

Parallel I/O, which enables distributed file access by partitioning datasets across multiple storage nodes, supports concurrent access by multiple tasks within a parallel application. It has been extensively explored in the literature [54–60] to enhance I/O performance. Parallel file systems offer a unified and transparent interface, enabling application programs to access files over the network as if they were on a local disk. Their support for concurrent, independent access enhances scalability and boosts I/O performance.

2.3.1 Parallel File Systems

The Parallel Virtual File System (PVFS) [54], now known as OrangeFS, as well as the BeeGFS (formerly known as FraunhoferFS) [60] are among the parallel file systems that will be considered in this thesis. OrangeFS is optimized for large-scale cluster computing, emphasizing high-performance access to large datasets in parallel applications [54]. It offers a consistent namespace throughout the cluster, supports user-controlled data striping across disks on separate I/O nodes, and facilitates existing binaries to use OrangeFS files without requiring recompilation [54]. As with many parallel systems, OrangeFS is a client-server system with multiple servers, called I/O daemons. These I/O daemons essentially run on separate nodes (referred to as I/O nodes) with disks attached. Each OrangeFS file is then striped across the disks on these I/O nodes. A client library facilitates application processes in interacting with OrangeFS and enables high-performance access through the Message Passing Interface (MPI). Additionally, OrangeFS includes a manager daemon responsible solely for metadata operations, such as permission checks for file creation, opening, closing, and removal [54]. The pvfs-client process enables the file system to be mounted and utilized with standard Unix utilities.

BeeGFS is a high-performance parallel file system initially developed at the Fraunhofer Center for High-Performance Computing, featuring a distributed metadata architecture. BeeGFS is designed to deliver the scalability and flexibility needed to support the most demanding HPC applications of today [60]. It offers a user-friendly integrated

graphical administration interface, automated cluster installation and setup, real-time usage statistics and health monitoring, a patchless client kernel module compatible with modern vanilla kernels, and built-in mirroring support. Figure 2.4 highlights a general layout of the BeeGFS which is typical of distributed parallel file systems.

Other distributed file systems include the Gluster [61] and Lustre [62] file systems. Lustre is designed for large-scale cluster computing, delivering high-performance file systems for small workgroup clusters to extensive, multi-site deployments. The Lustre file system is highly scalable, capable of supporting tens of thousands of cluster nodes with tens of petabytes (PB) of storage across hundreds of servers, and over a terabyte per second (TB/s) of aggregate I/O throughput.

Similarly, GlusterFS is a scale-out network-attached storage file system designed for high-performance computing. GlusterFS aggregates multiple storage servers using Ethernet or Infiniband RDMA interconnects to form a unified, large-scale parallel network file system. It consists of client and server components, where servers are deployed with one or more storage bricks (an abstraction of disks). Each server runs a glusterfsd daemon to manage and export these bricks, forming a volume that can be accessed by clients. The glusterfs client process communicates with servers using a custom protocol over TCP/IP, InfiniBand, or the Sockets Direct Protocol.

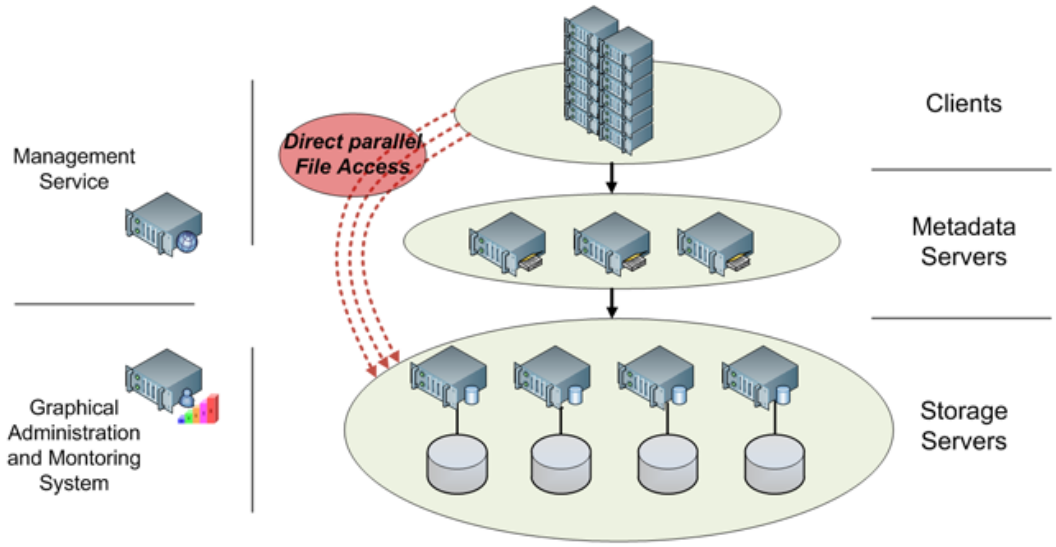


FIGURE 2.4: Internal system architecture of BeeGFS, illustrating the interaction between its core components, including the metadata servers, storage servers, and client interface. [60]

Despite the advances in I/O performance improvements by parallel I/O systems, data access in these disk-based data storage in parallel environments remains costly and poses challenges to application performance and throughput for parallel applications. A RAM-based storage system provides a much faster and efficient storage for applications, since accessing data from RAM is faster by several orders of magnitude ($\approx 1000\times$) [63] than accessing data from disk. Ousterhout et al. [63] proposed the RAMCloud concept, which is a DRAM-based storage system. Section 2.4.2 will provide a brief overview of the RAMCloud strategy as well as differences with EPGASS.

2.3.2 Parallel I/O Middleware

Parallel I/O is a technique that enables simultaneous data access on disk from multiple application processes to optimize bandwidth utilization. Accessing data in parallel from I/O can be a complicated process for the programmer. For instance, writing data in parallel while ensuring that the output does not suffer from race conditions may require explicit offset calculations or file locking semantics. To streamline this process, various parallel I/O middleware solutions abstract away the complexity from the application.

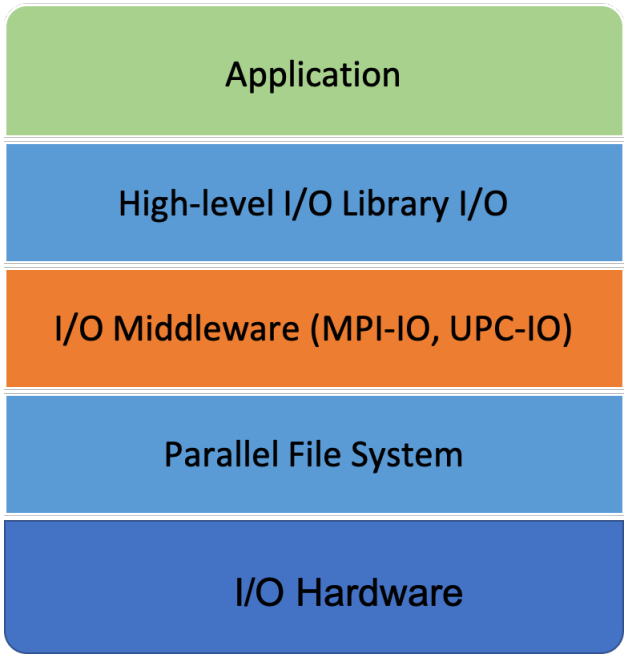


FIGURE 2.5: Parallel I/O Stack

MPI-I/O

MPI-I/O is a standardized and portable interface for parallel I/O, which is part of the MPI-2 standard, offering a low-level mechanism for performing parallel I/O operations [64, 65]. MPI-I/O includes features specifically tailored to meet the I/O requirements of parallel scientific applications efficiently. Positioned between the underlying parallel file system (which enables concurrent operations by MPI-I/O libraries) and the application or a high-level I/O library, MPI-I/O serves as an intermediary layer.

It is designed as an interface for a number of processes in a parallel (MPI) application to read from or write to different parts of a shared file. To achieve this, MPI-I/O implementations are typically built to leverage a parallel file system supporting shared access to a common file by multiple processes. High performance parallel I/O is made possible by allowing all processes to simultaneously access and modify distinct parts of the shared file.

MPI-I/O exposes POSIX-like interfaces for performing read/writes to files. For instance, the interface *MPI_File_open* takes an MPI communicator that specifies the group of processes that will access the file as the first argument, followed by the file name. After successfully opening the file, a process can access and read its designated portion of the data independently. MPI-I/O facilitates both **blocking** I/O, where read/write operations wait until completion, and **non-blocking** I/O, which allows overlapping I/O with computation or communication. ROMIO [66, 67] is a high-performance and portable implementation of MPI-I/O, that is specifically engineered for non-contiguous file access patterns that are frequently encountered in parallel applications. It is designed for compatibility with any MPI implementation and includes an optimized collective I/O mechanism. ROMIO serves as the foundation for many MPI-I/O implementations, variants of which are implemented in Cray, IBM's Blue Gene drivers, and OpenMPI MPI-I/O implementation. A generic sample code block for MPI-I/O operations is illustrated in listing 2.1.

```
MPI_File myfile;
MPI_Status mystatus;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
buffersize = FILESIZE / nprocs;
numints = buffersize / sizeof(int);
MPI_File_open(MPI_COMM_WORLD,
    ‘‘/mnt/pfs/datafile’’, MPI_MODE_RDONLY, MPI_INFO_NULL, &myfile);
MPI_File_seek(myfile, rank * buffersize, MPI_SEEK_SET);
MPI_File_read(myfile, buf, numints, MPI_INT, &mystatus);
MPI_File_close(&myfile);
```

LISTING 2.1: MPI-I/O Code fragment

Writing data in MPI-I/O can be performed in two modes: independent and collective I/O modes [65]. In the independent mode, each process writes to the file independently with no coordination from other processes in the program. In contrast, collective I/O involves gathering data at a few aggregator processes, which then write them to storage. The collective I/O mode is especially beneficial when there are many MPI processes, as it helps mitigate I/O performance degradation caused by excessive file system requests.

UPC - I/O

The UPC-I/O library is developed to integrate seamlessly with the UPC programming environment and builds on the foundations of MPI-I/O [68]. Unlike MPI-I/O, all UPC-I/O calls are collective and, thus, are invoked by all processes simultaneously [68]. The library facilitates both contiguous and non-contiguous data access. Collective tasks can be performed on shared or private buffers with individual or common file pointers. When a common file pointer is used, each thread is positioned in the file relative to that pointer based on their thread ID [52]. In contrast, with individual file pointers, each thread has its own pointer progressing through the file independently.

UPC-I/O inherently provides support for weak consistency as well as atomicity semantics. When data is written to storage by a thread, it is only visible to other threads

after all threads have closed the file collectively or synchronized their operations. These semantics ensure a balance between performance and correctness in parallel I/O environments.

2.3.3 The Adaptable I/O System (ADIOS)

ADIOS represents a multi-component and modular architecture I/O framework that delivers high I/O performance results on computer clusters. ADIOS has been demonstrated to deliver over a 1000× performance improvement over many well-known parallel file formats [69]. The ADIOS architecture uses a layered design approach to provide multiple I/O transport methods. This offers scientists the flexibility to choose the most suitable I/O method for various computing infrastructures with minimal modifications to existing applications, leveraging a suite of application programming interfaces (APIs). These APIs provide seamless methods to interface and use the underlying I/O of various computer systems. Rather than passing metadata as API arguments, all necessary metadata is stored in an external Extensible Markup Language (XML) configuration file. This ensures that the configuration is readable, editable, and portable across different machines. The latest version, ADIOS 2, offers unified application APIs with an abstraction layer designed to streamline data production and usage in scientific applications, thereby minimizing the cost of integrating various data transport technologies [70].

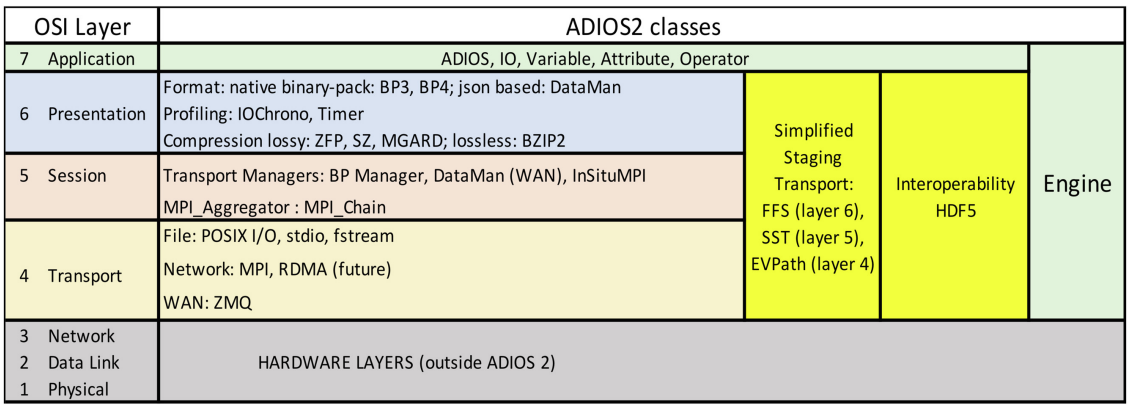


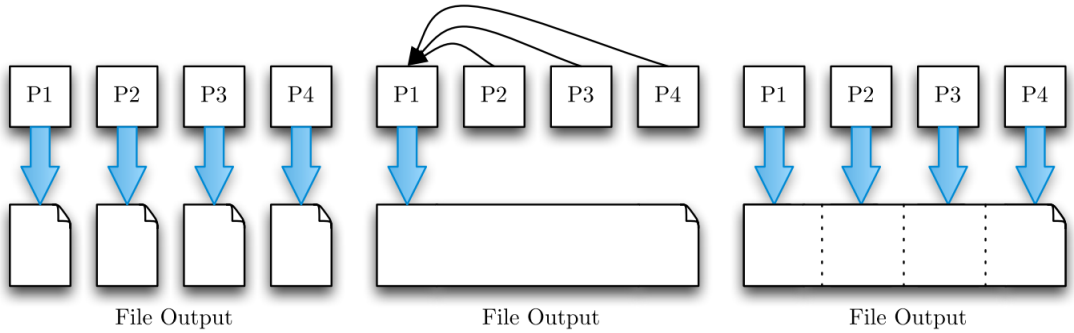
FIGURE 2.6: Basic layered design of ADIOS 2 highlighting how the internal architectural layers map to the standard Open Systems Interconnection (OSI) model abstraction [70]

2.3.4 Parallel I/O Challenges

Check-pointing or visualising data from a serial program may be relatively straightforward. However, it could be difficult to coordinate the writing and reading processes in a parallel application. This has led to many proposed strategies with different benefits and drawbacks. Figure 2.7 illustrates three (3) common approaches to handling data in parallel. In Figure 2.7a, each rank writes its own distinct data file; Figure 2.7b all processes send their data to a single "writer" ; and Figure 2.7c each process writes their data concurrently to the same file. Figure 2.7a usually achieves the fastest performance, but is the most difficult to manage. This could be the most effective approach if the program is consistently executed the same way with the same number of processes. However, data reload becomes computationally costly and complex if the application is launched on a different number of cores (e.g., initially run on P processors, writing M files, before data is reloaded from M files, but on N processors).

Figure 2.7b showcases a scenario in which the parent process serves as a standalone writer, consolidating the data into a single file as well as transferring across during a problem reload. Although this method is the simplest to manage, it is also the least efficient. The program benefits from increased parallelism, but the I/O process remains a serial bottleneck for the entire application run.

One of the most common approaches used in simulation programs is illustrated in Figure 2.7c. The method offers a balanced trade-off between performance and ease of management. It aligns with the design of many parallel file systems, and several communication libraries provide standardized APIs to facilitate how data is handled with this technique.



(a) N-to-N (b) N-to-root (c) N-to-1

FIGURE 2.7: Fundamental approaches for handling I/O in parallel applications

2.4 Distributed In-Memory Systems

In-memory computing is an approach to drive superior application performance by having large amounts of data reside in-memory, which allows for faster processing by various algorithms and frameworks. The overall goal of in-memory systems is to improve CPU – I/O performance. In-memory systems are, however, constrained by the amount of memory available on a single node, especially for large data processing tasks. Distributed memory parallel systems comprise multiple processing nodes connected by a high-speed network. Each node contributes its processing power and local memory for the overall processing of applications. In a distributed memory system without shared access, each processor can only access its local memory, with a messaging system that facilitates data transfer across the network between processors. In a shared system, however, each node has access to its local memory storage as well as remote access to portions of memory from other nodes. The distributed shared memory in this case provides a large memory pool for seamless large in-memory data processing for applications. Some examples of distributed in-memory systems include Alluxio, RAMCloud, and Memcached.

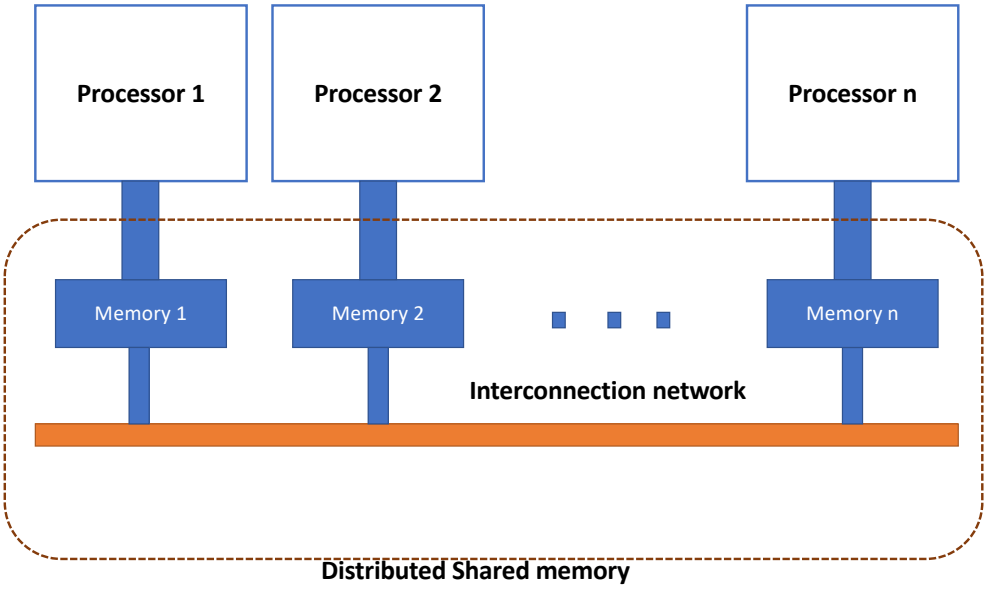


FIGURE 2.8: Diagram of the distributed shared memory abstraction, depicting the mapping of memory across multiple compute nodes into a single global address space accessible by all processes.

2.4.1 Alluxio: A Virtual Distributed File System

Alluxio is a virtual-distributed storage system that aims to bridge the disparity between computational frameworks and storage systems, allowing applications to access multiple storage infrastructure through a unified interface. It functions as a distributed shared caching service, enabling compute applications to transparently cache frequently accessed data, particularly from remote locations, to achieve in-memory I/O throughput. Furthermore, Alluxio's tiered storage, which utilizes both memory and disk (SSD/HDD), offers a cost-effective solution for elastically scaling data-driven applications [71]. Applications interact with Alluxio through a standard interface, accessing data from a unified source to ensure both high performance and seamless integration of applications. Furthermore, Alluxio adopts a memory-first tiered architecture, which enables data access at significant speeds [71].

2.4.2 The RAMCloud

RAMCloud is a storage system which stores data in the main memory of commodity servers and scales by utilizing hundreds or thousands of these servers to form a large-scale storage infrastructure [63]. As noted by Ousterhout, DRAM-based storage is not only faster than disk or flash for I/O-intensive workloads but also more cost-effective and energy-efficient. In RAMCloud, DRAM serves as the primary storage for data, while disk is utilized solely for persistence and backup. Ousterhout et al. [63] estimated that access latencies of $5\mu s$ to $10\mu s$ can be achieved end-to-end for an application server process reading a few hundred bytes of data over a network. The primary objective of RAMCloud is to aggregate DRAM from commodity servers to provide greater performance for data-intensive applications. Figure 2.9 depicts a typical architecture of RAMCloud.

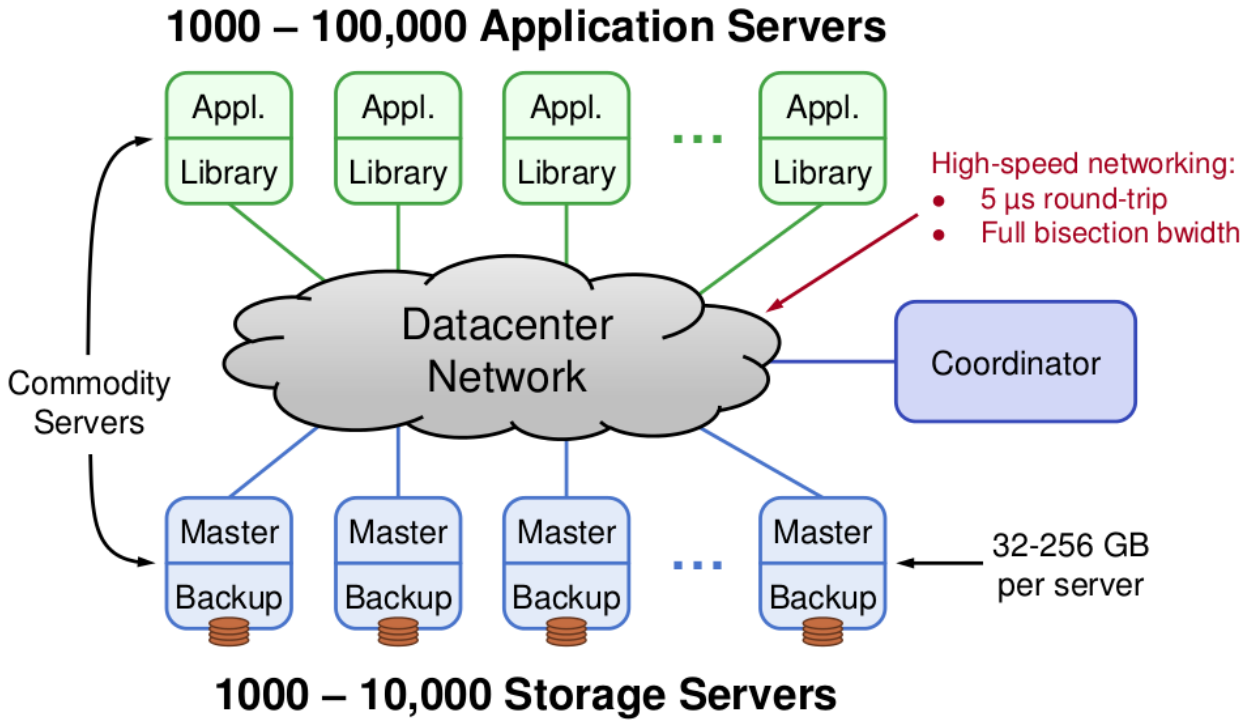


FIGURE 2.9: RAMCloud Architecture [63]

RAMCloud was motivated by the fact that if all data resides in RAM, applications can access them at memory speeds, and thus allow for complex data manipulation and analysis. While RAMCloud offers higher throughput for applications, it is specifically designed for cloud environments, focusing on aggregating the available RAM across

large clusters to host large amounts of online data, catering to the demands of large-scale web applications [31]. The objective is to deliver efficient RPC APIs and high I/O performance through the use of specialized protocols and optimized network infrastructures [31]. Luo et al. [31] noted that achieving low round-trip network latency on the scale of microseconds may require costly InfiniBand switches. For most data centres using Ethernet/TCP/IP-based network infrastructures, deploying such a RAMCloud system to achieve higher throughput would be challenging. Furthermore, it is not clear how RAMCloud will perform in the face of increasing complex HPC applications and scientific computations.

2.4.3 Memcached

Memcached is an open-source system for distributed memory object caching. Though designed primarily for web applications, Memcached could be used as a general-purpose caching system. Memcached caches data and objects in memory to reduce the need for frequent access to external databases or APIs, thereby improving application performance and efficiency. In the Memcached system, each item consists of a key, an expiration time, optional flags, and raw data [72]. When a data request is made, Memcached verifies its expiration time to ensure validity before returning the requested data. By default, Memcached uses the Least Recently Used (LRU) cache with expiration timeouts. When the server runs out of memory, it first replaces expired items. If additional memory is still required, Memcached evicts data items that have not been accessed for a specified duration (equal to or longer than the expiration timeout), prioritizing the retention of recently accessed information in memory for quick retrieval.

Memcached has four basic components:

- Client software, which gets the list of available Memcached servers.
- A client-based hashing algorithm: this chooses a server based on the key input.
- Server software, storing the data values with their keys in an internal hash table.
- Server algorithms, which define when to discard old data or reuse memory.

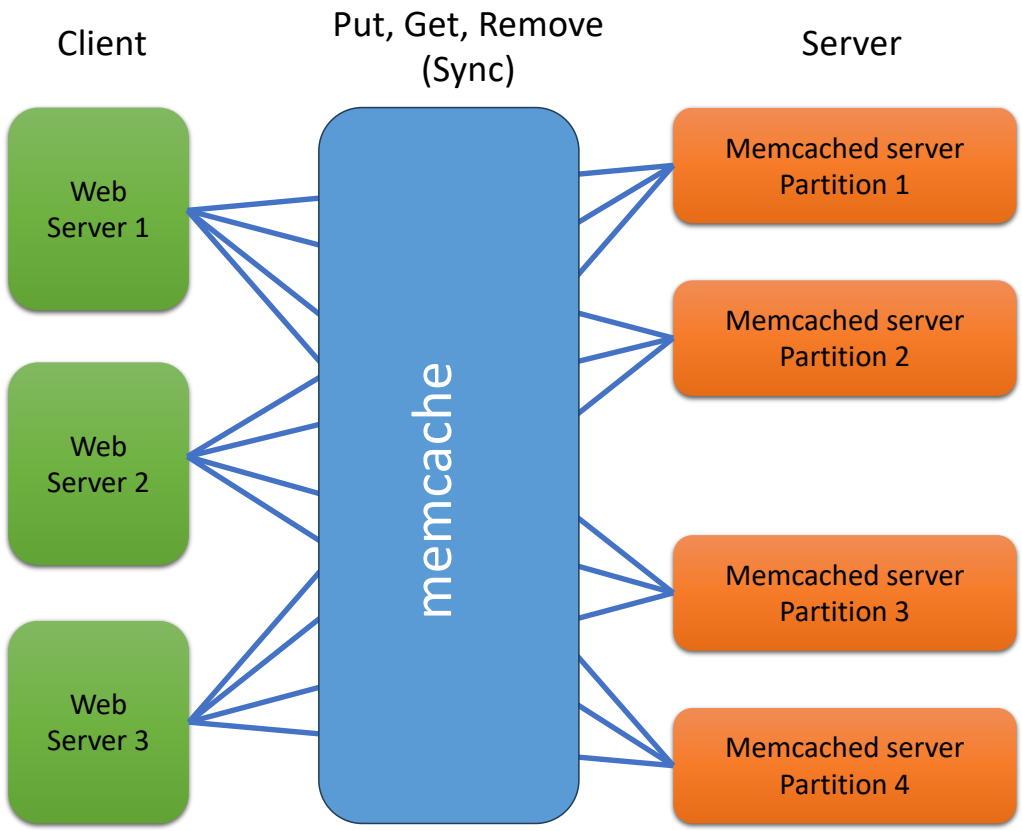


FIGURE 2.10: Sample Memcached setup

2.4.4 DataSpaces

DataSpaces is a data management framework designed for integrated program workflows on large-scale systems. It is built to facilitate dynamic engagement and coordination patterns among scientific applications, offering a semantically specialized shared-space abstraction through a set of staging nodes. DataSpaces offers services such as a distributed in-memory associative object store, scalable messaging, and runtime mapping and scheduling for online data analysis operations [33, 32]. It enhances overall data access performance by leveraging memory layers across distributed computing nodes to enable in-memory persistent data storage while managing data placement across server nodes and various storage tiers, including DRAM, NVRAM, and SSD. It has been integrated with the ADIOS2 framework, which is distributed by Oak Ridge National Laboratories. The current layered architecture of DataSpaces uses Mercury as the communication layer. Mercury is a remote procedure call (RPC) framework tailored for HPC systems that utilize high-performance network fabrics. The implementation of

the network is abstracted to optimize the use of native transports and facilitate seamless portability to a wide range of systems, including future platforms [73].

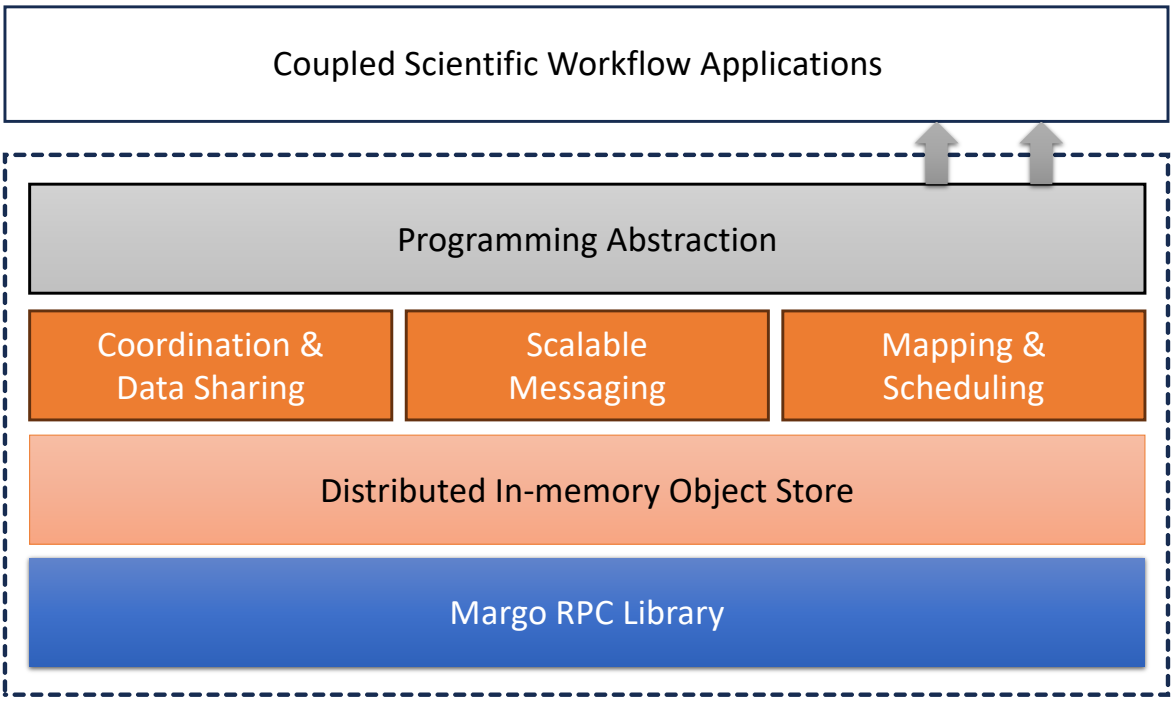


FIGURE 2.11: DataSpaces architecture showing the structured layers and components – the distributed in-memory associative object store, scalable messaging, as well as the runtime mapping and scheduling.

2.5 Memory mapped file I/O

Memory-mapped file I/O is another technique which allows efficient work with a large file, without mapping the whole file into memory. In modern operating systems, memory-mapped I/O (MMIO) is a key access technique that associates a file or file-like resource with a segment of memory [74]. This memory-map allows applications to interact with the files through memory semantics. The advent of low-latency Non-Volatile Memory (NVM) storage has brought renewed attention to memory-mapped file I/O as a method to reduce software overhead and achieve high throughput. A segment of virtual memory is directly mapped to a corresponding byte-for-byte portion of a file or a file-like resource. Using the *mmap* system call to map a file onto the user memory space allows users to access the file directly as if it were data in memory [75]. This approach eliminates the need for a complex I/O stack in existing operating systems and avoids user/kernel mode switches associated with read/write system calls.

As a result, user/kernel mode switching is significantly reduced. Furthermore, no data copying is required between the kernel space and the user space, thus minimizing overhead. For these reasons, the *mmap* system call is likely to become a pivotal interface in the future [76, 74]

2.6 In-memory Databases

High performance computing infrastructures enable large-scale scientific applications, analytics, and work-flows to run with increased complexity, as well as improved accuracy and granularity. With the explosion of data and improvement of in-memory data technology, there is a major shift to in-memory data processing; driving several research to strategically leverage in-memory computing to effectively process and access data from massive stores. Advances in parallel computing and CPU speeds which have further widened the CPU–I/O speed gap have further motivated the use of in-memory computing in many aspects. Apache Ignite [77], for example, is a distributed Database for High-Performance Computing designed to process databases with in-memory speed. It uses a multi-tier storage system including memory, and scales horizontally across memory and disk tiers, providing for fast data accesses. Other vendor developments leveraging in-memory computing include: Oracle's Timesten [21] and SAP-HANA [24]. In-memory computing and data processing provides the means to store and process data in RAM distributed across clusters of computers, providing for high-speed data accesses. This high-performance architecture is rapidly emerging as the preferred approach for modern big-data and fast-data applications.

2.7 Summary

In this chapter, the thesis explores some of the various techniques and tools that are being used today to drive the high-performance and in-memory data processing agenda, including MPI for message passing, UPC for distributed shared-memory programming, and their respective configurations for high throughput and computational efficiency. The chapter further discusses distributed in-memory systems such as Alluxio, Memcached, DataSpaces and the RAMCloud which have been developed to leverage

in-memory computing, and improve computational efficiency for diverse applications and systems. Although RAMCloud has been developed to enable in-memory data processing, it focuses on large-scale web applications rather than complex scientific computations. Additionally, while message passing is one of the most popular paradigms for parallel programming, it does not provide a global address space, which limits its ability to handle dynamic, irregular data structures (e.g., trees) efficiently across processes. In addition, the need for explicit communication between processes could lead to potential error-prone code, especially in complex applications with many communication patterns.

The thesis proposes the EPGASS architecture which provides a similar concept as the RAMCloud for in-memory computing, but uses industry standard HPC tools to provide a distributed DRAM-based storage for scientific application and big-data analytics leveraging commodity hardware and low-cost network. This study considers the use of 10 Gigabit Ethernet (GbE) switch instead with RDMA over Converged Ethernet (RoCE) or Internet Wide Area RDMA Protocol (iWARP) with an Ethernet network. A 10 GbE Ethernet provides a cost-effective network, while leveraging RDMA for Remote Memory access (RMA) and provides ease of use for existing scientific applications. However, when Infiniband is used, the system effectively leverages the use of APIs for RDMA. The details of the EPGASS system architecture including software stack and design goals are discussed in Chapter 3.

Chapter 3

The Elastic Partitioned Global Address Space System

3.1 Introduction

The Elastic Partitioned Global Address Space Storage *(EPGASS)* is an in-memory data storage system based on the Partitioned Global Address Space model of computation. The architecture is made up of a cluster of nodes using commodity hardware, in which a number of them are configured as data storage nodes, while the others are configured as compute nodes. Large portions of the RAM segments of the storage nodes are aggregated to constitute a shared logical global address space. The system is *elastic* in that the number of storage nodes could be increased or decreased (i.e. dynamically adjusted) for different sessions of application runs depending on the needs of the application at any time.

3.2 Architectural Overview

This section discusses the system architecture of EPGASS and the rationale behind the design choices. The general operational architecture is depicted in Figure 3.1. The basic components include:

1. *I/O Nodes*, each of which has a large DRAM of the order of 10 $\sim$ 100 GBs

2. *Compute Nodes*

3. *High-Speed Network Interconnects* (Infiniband or 10 GbE) ;

4. *Solid-State Drives* (SSDs)

5. *Large capacity disks*

In HPC systems, the technique of decoupling compute and storage enhances processing efficiency and overall application throughput. The EPGASS architecture allows improved allocation and utilization of storage and compute resources, while ensuring cost savings with the use of commodity hardware. The two major challenges with this architectural design are the performance impact due to lower bandwidth and higher latency, especially access to remote storage, and the limitations of creating or modifying an application against a particular storage interface. EPGASS mitigates the performance impact of remote storage by managing the storage media of the computation cluster and ensuring hot data locality for applications in memory.

The *ADIOS* system is adopted to ensure that applications can interact with any underlying storage interface without the need to directly implement storage-specific semantics. Thus, this provides seamless integration and application usage for various storage systems. The functions and features of EPGASS are further augmented with the high-performance data transfers with DataSpaces [33, 32].

Figure 3.2 presents the software system stack for the EPGASS setup, depicting the major components leveraged in the architecture, including UPC for elastic memory abstraction, ADIOS for I/O middleware, and MPI-IO/UPC-IO for interfacing with the parallel file system.

3.2.1 Input/Output Nodes

The I/O nodes are configured with a parallel file system and used for persistent and archival storage purposes. They provide storage services as well as interface between compute and storage service nodes. The storage nodes implement a hierarchical storage scheme including main memory (DRAM), fast solid-state drives (SSDs) and then

slower disk (HDD) for archival storage. Storage systems are a key component for data-intensive computing tasks such as large-scale scientific simulations, experimental or observational science, as well as many machine learning applications, which have gained a lot of attention in recent times. Many of these applications have seen an increasing need to process large volumes of data between tasks, typically achieved through the storage and I/O system. Additionally, certain workflows may involve reading substantial amounts of input data, such as initial conditions for starting a simulation, boundary conditions for each simulation step, large training datasets, or Experimental and Observational Data (EOD) for comparison with simulation results or other purposes [78]. Efficient and quick access is key in all such cases to ensure a smooth and effective solution. Slow storage I/O will have a negative impact on the overall performance of an algorithm or simulation. Some applications, however, may require only a modest amount of input data at the start of a simulation, along with additional data when resuming from a previous run. This input data generally consists of parameters that establish the initial conditions for the simulation, which may be utilized in the differential equations that dictate the evolution of the simulated variables. In such scenarios, the input data may be shared among the processors. The EPGASS architecture ensures this initial data is loaded and stored in memory together with other outputs which will be referenced as the application proceeds. Outputs from applications are also persisted to the storage system I/O (SSIO). Using immutable storage specifically for input files can reduce I/O overhead and enhance overall application performance. Additionally, as simulations increasingly validate their results against experimental or observational data, it becomes necessary to read data from various experiments to ensure the simulation accurately reflects real-world conditions. For instance, in many fusion experiments, data from numerous diagnostic instruments on fusion devices are ingested at the start of a simulation. Over time, as fusion reactors expand and incorporate more diagnostic instruments, each capable of collecting data at faster rates, the volume of data to be processed continues to grow.

Simulations generate various types of output data, typically categorized into two groups: defensive output for error recovery and productive output for scientific purposes. Defensive output often includes global checkpoint files, which store a consistent snapshot

of the simulation state on persistent storage, enabling the application to restart in case of premature termination. Productive output, on the other hand, consists of data representing the current state of the simulation for scientific analysis. Often, defensive output files also serve as productive output, as all data (e.g., from a combustion simulation like S3D [79]) may be required for comprehensive data analysis. Checkpoint files, which contain all the information required to fully reconstruct the simulation's state, are typically larger than productive analysis output, which only summarizes key features of the simulation. Consequently, as the data collected from experiments increases, the volume of data passed to simulations grows proportionally [78].

In applications where checkpoint files are used for analysis, some simulations can generate petabytes of data in a single run, with future runs potentially producing numerous such files, collectively reaching exabyte scales. For cases where checkpoint data serves both productive and defensive purposes, users often adjust the check-pointing frequency based on analysis requirements rather than error recovery needs. This often results in more frequent check-pointing than the "optimal" rate recommended for error recovery [80]. Additionally, application scientists may modify the check-pointing frequency to ensure that I/O time remains a relatively small fraction of the total execution time.

As parallel computing systems expand, there is increasing interest in shifting away from writing checkpoints to global parallel file systems. Hybrid checkpoint schemes, such as the Scalable Checkpoint/Restart (SCR) library [81] and the Fault-Tolerant Interface (FTI) [82], have gained significant traction and widespread acceptance among application scientists. EPGASS leverages the Berkeley Lab's Checkpoint Restart (BLCR) [83] to provide seamless check-pointing and restart for applications.

The EPGASS I/O nodes include a range of commodity hardware, leveraging state-of-the-art software such as low-level parallel file systems (OrangeFS, Lustre, or general parallel file system [GPFS]), to provide improved capability, scalability, and robustness which allow for larger volumes of data storage and processing as the needs of the application work-flow may require. Leveraging a well-supported high-level I/O

library enables efficient data sharing within a large scientific community [78]. Professional software development efforts can focus on creating high-quality data analysis tools utilizing these I/O libraries. For instance, the climate science community utilizes a wide range of data analysis tools to process petabytes of NetCDF files [84]; the high-energy physics community employs an efficient data analysis framework based on ROOT files [85]; and researchers in the fusion community frequently use ADIOS-BP to exchange vast amounts of simulation data, amounting to many petabytes [56]. These shared I/O libraries simplify the process of reading and writing large volumes of data in parallel. EPGASS utilizes the Adaptable I/O System (ADIOS) [70, 69], offering a straightforward and flexible method to describe data within code that may need to be written, read, or processed externally from a running simulation or computation.

3.2.2 Compute Node

Compute nodes constitute the majority of the system and act as the primary workhorses, delivering computation for applications. Compute nodes are responsible for acting on data/tasks that are submitted and scheduled via I/O nodes. The data being processed and the results of the processing are stored on the storage nodes. Modern commodity hardware comes equipped with non-volatile random access memory (NVRAM) that can be used for storing checkpoint files and frequently used input data (e.g., through systems like DataWarp [86]. NVRAM enables faster writing of checkpoint data, reduces traffic to slower disk storage systems, and thereby enhances I/O efficiency and overall application performance.

EPGASS conducts all computations using the aggregated distributed memory of these compute nodes, including storing check-pointed files for recovery and fault tolerance. Each storage node is also complemented with SSDs of reasonable capacity to temporarily store memory resident data from applications should memory data become considerable to fit all into distributed memory during an application run. The contents of each RAM are temporarily persisted onto the SSDs attached to the nodes. These SSDs serve as intermediary fast storage system for check-pointing and running programs. An alternate configuration would involve using SSDs as the permanent persistent storage;

however, due to limitation in capacity and expense of SSDs compared to HDDs, using them as interminably storage to improve performance of persistent writes and fast data loading provides a cost-effective solution to the problem of persistent and data durability.

The Berkeley Lab's Checkpoint/Restart (BLCR) is used for the purposes of check-pointing applications and ensuring resilience and fault-tolerance. The in-memory data are stored persistently and made durable asynchronously during application runs.

The BLCR provides the following functionalities [83]

- *Scheduling*: Check-pointing enables a program to be paused at any point during execution, allowing another program to run in its place. The original program can then be resumed at a later time.
- *Process Migration*: If a compute node is at risk of crashing or needs to be shut down for reasons such as routine maintenance or hardware upgrades, checkpoint/restart enables processes running on it to be migrated to another node or saved for resumption once the original node becomes available.
- *Failure recovery*: A long-running program can be periodically check-pointed, allowing it to restart from a more recent execution point rather than starting over if it crashes due to hardware failures, system software issues, or other non-deterministic causes.

EPGASS integrates and leverages cluster state-of-the-art resource managers such as OpenPBS, which optimizes job scheduling and workload management, while ensuring system efficiency and user's productivity. This ensures fast scheduling, scalability, security, and resiliency, for the HPC infrastructure, middleware, and the software application stack.

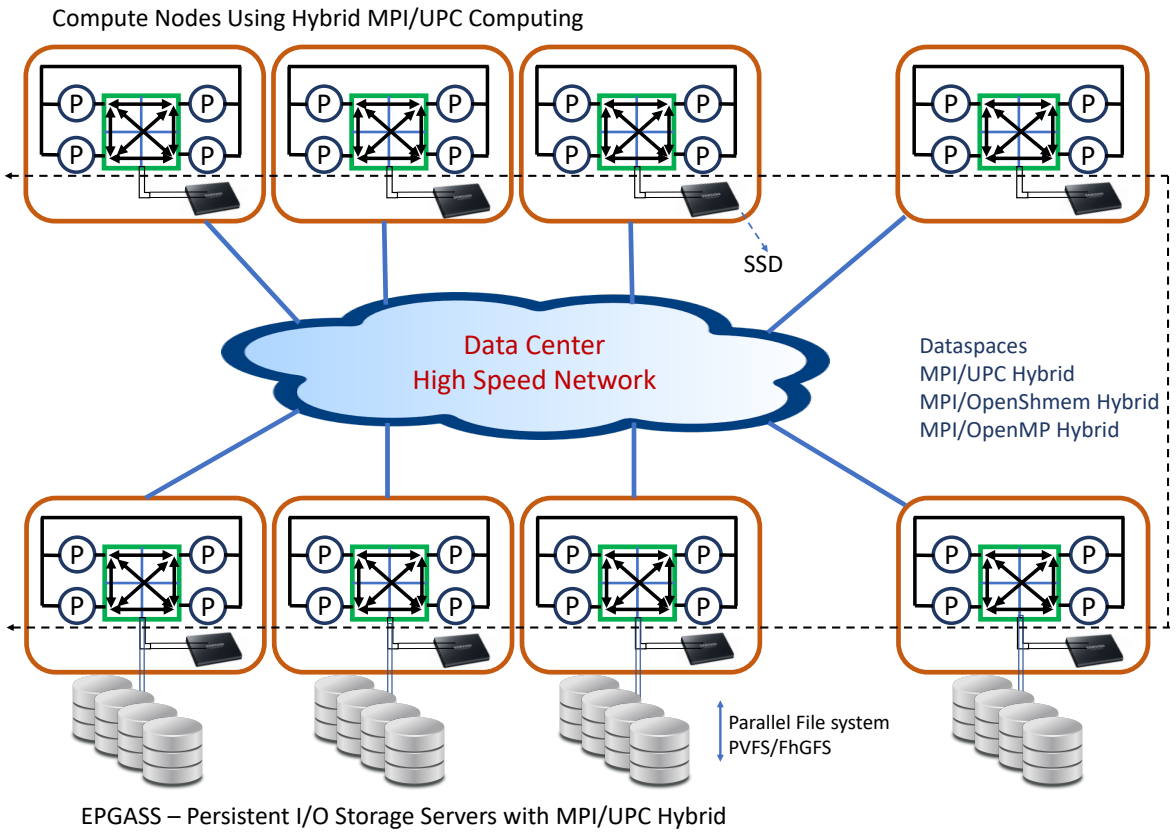


FIGURE 3.1: Schematic Diagram of the Architecture of EPGASS highlighting the compute and I/O nodes setup with commodity hardware which leverages PGAS model for in-memory application processing and data storage

3.2.3 High-Speed Network Interconnects

High-speed interconnects are the backbone to HPC systems and a major contributor to the overall performance. Communication plays a pivotal role in HPC applications. It is required to gather results of a computation or distribute tasks among threads on different or same node. HPC networks continue to improve significantly in performance, and thus challenge system designers to provide storage performance capable of efficiently leveraging and saturating the capacities available on the local network interfaces. Modern HPC interconnects achieve latencies of about 90 ns to 1μs, with data throughputs reaching several hundreds of Gbps [87–89]. The ability to aggregate tens of GB/s of network bandwidth into a single storage node further provides increased capacity, which requires a matching amount of storage bandwidth to produce an economically efficient storage system. A significant percentage of HPC application time

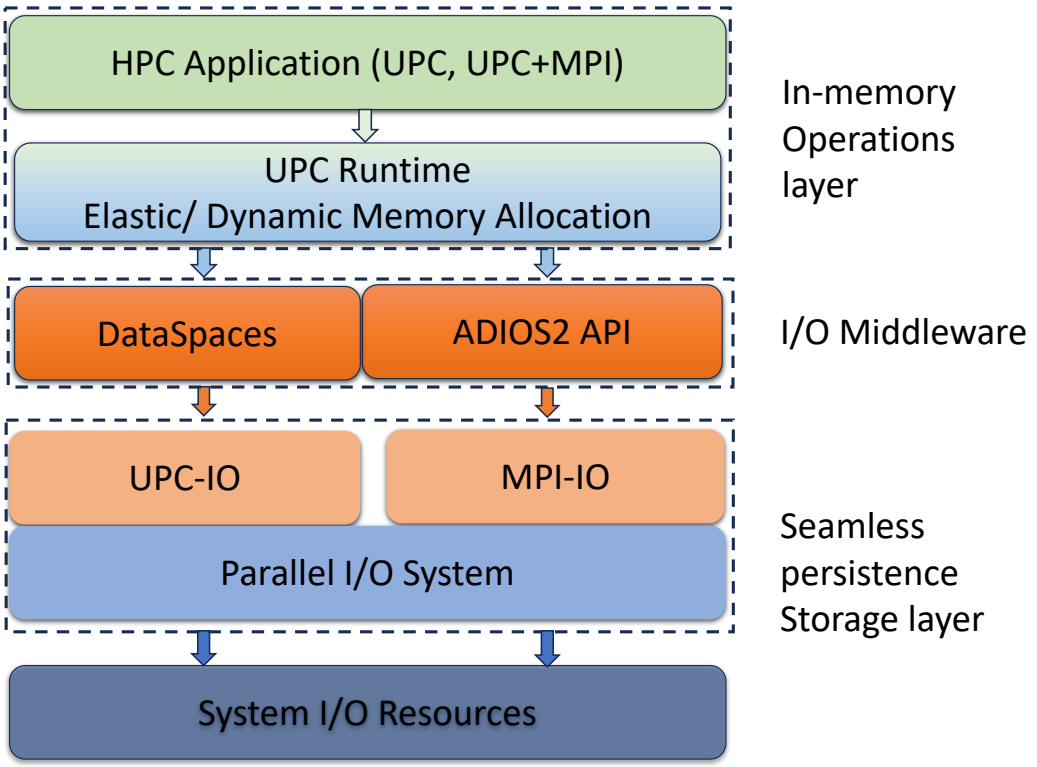


FIGURE 3.2: Software system stack of EPGASS highlighting the various systems leveraged for applications and data processing

is spent on communication, especially in large scale data-intensive HPC applications. Communication could take up to 55% or more of the total running time in such applications [90]. Slow communication network, especially internode communication, significantly reduces the overall application performance. Advances in networking that support hardware offloading of communications and allow CPUs to be fully dedicated to computational tasks while communications are independently managed by network hardware significantly reduce communication overhead in applications, ensuring increased overall performance gains.

There are several HPC interconnects that implement varying technologies and schemes to enhance performance. Today, some of the most powerful supercomputers are composed of clusters using high-speed InfiniBand connectivity. EPGASS uses a high-speed *10/25* GbE network fabric with iWARP (Internet Wide-Area RDMA Protocol) support. iWARP (Internet Wide Area RDMA Protocol) is a network protocol designed to enable RDMA over IP networks [91]. RDMA enables direct memory access between nodes

without requiring intervention from the operating system on either node (host bypass). This technology facilitates high-throughput, low-latency networking with minimal CPU usage, making it particularly advantageous for massively parallel compute clusters. RDMA allows direct memory-to-memory data communication between applications over the network at high speeds. With remote direct-memory data access, inter-node network communication overhead is significantly reduced to as low as 1–2μs — enhancing overall application performance.

3.3 Hierarchical Storage Structure

EPGASS leverages a 3-tier storage hierarchy (Levels 1,2 and 3 illustrated in Figure 3.3) to optimise storage performance, capacity, and cost. Level 1 consists of high-speed distributed system RAM which facilitates fast reads and writes. Large-capacity SSDs make up the storage devices in Level 2. These fast SSDs improve performance of applications with very large memory footprint by acting as large capacity caches for each node. In-memory data is asynchronously flushed to the SSDs, which provide faster access to the storage I/O infrastructure. Traditional large capacity disk drives, which constitute the Level 3 layer, are the cheapest and much slower, and therefore used mostly for archival purposes where necessary. For instance, for a simulation experiment, all computations are conducted in-memory with data distributed across the various compute nodes. Check-point files from such a simulation are stored in memory and asynchronously persisted to SSDs as the simulation proceeds. The checkpoint files are retrieved from the SSDs only when the memory resident data is unavailable and the system needs to restore experiment to previous state. Simulation results are also persisted to SSDs and subsequently to the slower large capacity disks.

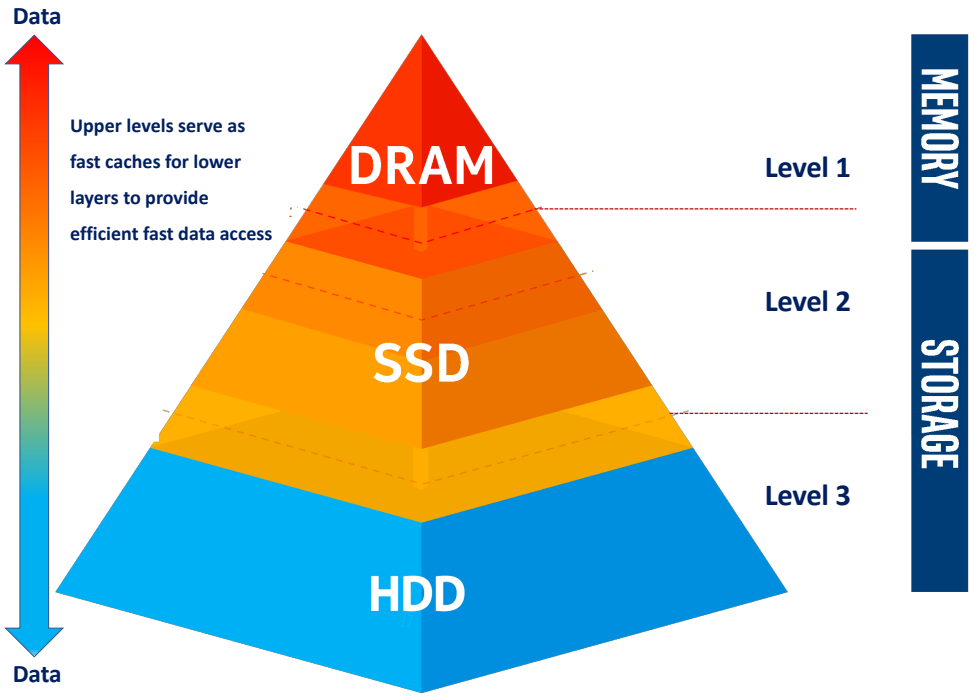


FIGURE 3.3: Hierarchical Storage Structure

3.4 Fault Tolerance And Recovery

Giving that large number of nodes and components interact with each other in large scale HPC systems, faults are unavoidable. It is crucial for a system to recover from faults transparently, without significantly impacting performance or compromising other features. The system must be able to recover from failures as fast as possible with minimal impact on the normal application runs. EPGASS ensures low latency and fast recovery by leveraging distributed data storage on fast SSDs. Data persisted to SSDs are replicated asynchronously across multiple nodes while being persisted for archival storage. In addition, the application of replication strategies such as Tailwind [92] further provides a substantial improvement in replication throughput and response latency. Tailwind is a zero-copy recovery-log replication protocol for scale-out in-memory databases, leveraging one-sided RDMA [92]. This ensures fast and efficient data retrieval from storage needed to resume the application run.

3.5 Summary of EPGASS Architecture

The shift toward many-core architectures increases the load and contention on the I/O stack, posing significant performance challenges for I/O in parallel scientific applications. The generation, analysis, and management of numerous extreme-scale datasets, combined with the advent of new storage and networking technologies, place additional strain on the storage capabilities of current HPC systems. Many large-scale scientific experiments routinely write out immense amounts of petabytes of data on today's HPC systems. Such "big data" imposes steadily increasing pressure on the storage and I/O systems. However, applications achieve significant performance gains when disk I/O is minimised. I/O is widely recognized as a key performance bottleneck for both large-scale scientific applications and data analysis, and this is expected to worsen with increase in the disparity between computation and I/O capacity on emerging exascale machines. Minimising I/O and leveraging distributed memory storage during computation with the EPGASS strategy, thus provides significant performance improvement and gains.

The basic principle behind EPGASS architecture is the use of commodity hardware and systems leveraging in-memory computing (IMC) to provide high performance I/O instead of specialised and vendor-locked systems which could be difficult to manage, and expensive to maintain in the long term. The architecture achieves high performance by ensuring significant use of memory resident compute, leveraging the partitioned global address space paradigm. Dedicated compute nodes configured with large DRAMs or NVRAMs, as well as SSDs; Storage I/O nodes enhanced with a hierarchical storage stack including a parallel file system and a high bandwidth interconnect between nodes to provide performance enhancement especially for applications with large memory footprint. Application checkpoint data is stored in-memory during application run and asynchronously persisted through a hierarchical storage structure: which includes high-speed memory to fast SSD and subsequently to relatively slower disks where applicable. With EPGASS, applications with high memory footprint and supporting high parallelism could achieve high throughput and performance gains by minimising I/O usage during active computation.

Chapter 4

EPGASS I/O Performance Benchmark and Analysis

4.1 Introduction

The EPGASS system architecture aims to provide low latencies and high-speed fault-tolerant data access in a distributed shared-memory storage infrastructure leveraging the partitioned global address space model of computing and its high-performance Global Address Space Networking (GASNet) library. This facilitates support for one-sided, remote-memory-access (RMA) communication (i.e., *Puts* and *Gets* being used RAM distributed physically across nodes), as well as sensitivity to the latency and overheads of fine-grained communication. The GASNet API supports two main modes of communication: (1) a one-sided RMA which leverages the RDMA capabilities of underlining network hardware, enabling their use to directly implement PGAS *Put*/*Get* procedures on user data structures, and (2) a tailored Active Message (AM) interface to allow extensibility and efficient management of the parallel runtime system. This study established a minimal proof-of-concept infrastructure and conducted benchmarking experiments to validate the system.

The study set up a twelve (12) node cluster codename "Jaguar cluster" with commodity 8-core (2 threads/core) 2.40GHz Intel Xeon E5-2630 v3 processors each with 16GB of RAM and 2 TB SATA HDD, running the Ubuntu 16.04 LTS (Xenial Xerus) operating system. Each node was configured with 32GB swap storage, as well as 500GB of

storage space designated for data storage space for parallel file system. Each data node was also augmented with 256GB Samsung 860 EVO SATA 3 SSDs to provide faster data access in a hierarchical access pattern. The cluster nodes were interconnected by a simple, reliable, and high-performing star network topology using a 10GB Ethernet switch connected to 10GB ports attached to Peripheral Component Interconnect Express (PCIe) interfaces on each node.

The study set up EPGASS on the cluster leveraging various state-of-the-art HPC tools such as MPICH2 with ROMIO: a high-performance portable MPI-I/O implementation. ROMIO is optimized for non-contiguous access patterns, which are common in parallel applications. It has an optimized implementation of collective I/O which is an important optimization requirement for parallel I/O. Additionally, the Fault-Tolerant Backplane (FTB) was enabled, including support for distributed memory access with the Berkeley Unified Parallel C (UPC) with remote memory access across nodes. For this setup, the OrangeFS parallel file system was configured to provide scalable and high-throughput I/O for the cluster.

The *jaguar01* node, which is accessible from the *local* network and the internet, served as the gateway to the cluster nodes and executed the development tools. Each host mounts its /home directory from the Network File System (NFS) running on *jaguar01*, making it easy to build on *jaguar01* and execute the resulting binaries on *jaguar02* through *jaguar12*. Each host is also configured with partitions /data0 and /data1 dedicated to archival storage partitions used by the parallel file system configured on the nodes. To set up the cluster for the various benchmarking experiments, *jaguar02* to *jaguar06* were configured as compute nodes, while *jaguar07* to *jaguar12* were configured as storage I/O nodes. Experiments were run on the nodes to characterise the cluster's performance under various workloads.

4.2 Benchmarking Tools

The study conducted various benchmarking tests using a series of benchmarking tools on the storage infrastructure to experimentally analyse the performance of the storage

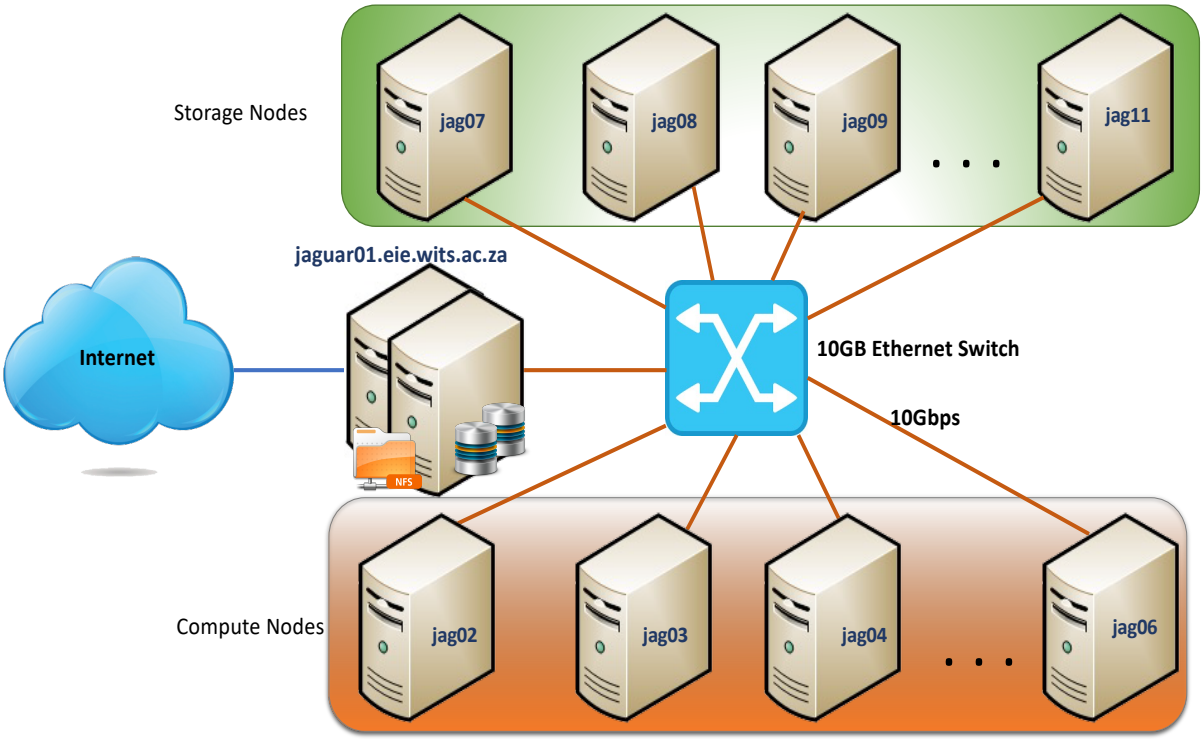


FIGURE 4.1: Layout of testing cluster codenamed "Jaguar"

infrastructure and data storage concept. The benchmarking tools and their respective metrics are discussed in the following sections.

4.2.1 The NAS Parallel Benchmarks

The NASA Advanced Supercomputing (NAS) Parallel Benchmark (NPB) is a suite of programs initially implemented by NASA's Advanced Supercomputing division to characterize the performance of parallel computing systems [93]. The consolidated suite (largely drawn from computational fluid dynamics (CFD) applications) contains a number of programs designed to facilitate assessing the operating efficiency and performance of parallel systems. The application is made up of different programs and pseudo-programs with essential features such as *intensive memory communications, complex data dependencies, different memory access patterns* and *hardware components/sub-systems overload* [94]. The suite also supports tests for unstructured adaptive meshes,

parallel I/O, multi-zone programs, and grid-based computations.

The NPB suite contains kernels and simulated programs that represent different workloads. These workloads present different performance tests, including random memory accesses, data dependencies with diverse complexity, as well as short-and long-range data communications [93], providing characterization for data locality and memory bandwidth capacity. The kernels from the benchmark suite include:

- Embarrassingly Parallel (**EP**). This kernel constructs a high number of Gaussian random deviates and processes them using the acceptance-rejection method to compute the Gaussian deviation. It then calculates the number of pairs that fall within the square annulus. This approach is effective for evaluating the system's floating-point operation capacity.

- Multi Grid (**MG**). This kernel employs the V-cycle Multigrid approach to solve a 3D scalar Poisson equation, iteratively performing restriction and prolongation as it alternates between coarse and fine grids. The objective is to test both long-distance and short-distance data communication.

- Conjugate Gradient (**CG**). It approximates the smallest eigenvalue of a large, sparse, and unstructured matrix using the Conjugate Gradient approach. This kernel targets the performance of data communication mechanisms, memory locality, and cache utilization.

- Discrete 3D Fast Fourier Transform (**FT**). This kernel performs a Fast Fourier Transform (FFT) on a 3D partial differential equation using spectral and inverse methods within an iterative loop, simulating intensive long-distance communication [93].

- Integer Sort (**IS**). The *IS* kernel executes integer sorting on a sparse dataset, simulating a critical computation for particle-in-cell applications. It employs the Bucket-Sorting algorithm by default and evaluates integer computation and data communication performance.

On the other hand, the pseudo-applications in the benchmark suite provide three (3) different iterative algorithms to evaluate a 3D Navier–Stokes system of differential equations which describe the flow of incompressible fluids. They are:

- Block Tri-diagonal solver (**BT**). This kernel employs a computationally intensive implicit algorithm to solve 3D Navier–Stokes equations numerically. The solution utilizes Alternating Direction Implicit (ADI) factorization on a 3D matrix, generating block tri-diagonal systems that resolve the unknown vectors along each direction through the back substitution method [93].

- Scalar Penta-diagonal solver (**SP**). The Beam-Warming approximate factorization is used to decompose the 3D matrix, resulting in a specific type of band matrices known as Scalar Penta-diagonal matrices. The Tri-diagonal Matrix Algorithm is then applied to the block-tridiagonal systems, followed by the back substitution method to solve the remaining vectors [93].

- Lower-Upper Gauss–Seidel solver (**LU**). It employs the Symmetric Successive Over-Relaxation (SSOR) method, which combines two SOR computations. SOR, a variant of the Gauss–Seidel method, is used to solve linear system of equations. The process involves a forward SOR sweep to update the variables, followed by a backward SOR sweep to update the unknown variables in reverse order.

The study employs the UPC NAS benchmark suite that was developed by the UPC group at the George Washington University [95] which is part of the Berkeley UPC setup. The UPC NPB has the kernels CG, EP, FT, IS, MG, and BTIO (Test of different parallel I/O techniques) present. Every benchmark within this suite consists of a "naive" implementation and one or more "optimized" implementations. The "naive" implementation makes no effort to implement any hand-tuning techniques, whereas the optimized implementations apply various hand-tuning techniques such as pre-fetching and privatized pointers-to-shared to boost performance.

The NPB benchmark offers different classes: S, W, A, B or C (smaller to larger sizes), which indicate the problem size for each benchmark experiment. These classes differ

mainly in the size of the problem, including much larger problem size classes; D, E, F: which are $16\times$ size increase from each of the previous classes (A, B, or C).

Table 4.1 shows the problem sizes and performance rates (measured in Mflop/s) for each of the eight benchmarks, for the classes A (4.1a), B (4.1b) and C (4.1c) problem sets.

4.2.2 The UPC STREAM micro-benchmark

The UPC STREAM benchmark [96] is a micro-benchmark suite which evaluates the sustainable memory bandwidths of a wide range of shared memory fine- and coarse-grained accesses. Generally, the performance of UPC and HPC programs is often significantly influenced by the cost of fine-grained remote memory references. The cost of coarse-grained references involves calls to *upc_memget*, *upc_memput*, *upc_memcpy*, and *upc_memset* routines, providing insight into the efficiency of the implementation of these message-passing operations. The augmented UPC STREAM synthetic micro-benchmark implementation, proposed by Zhang et al. [97], was employed in this study to simulate a broader range of access patterns. The original implementation is limited to local shared references and remote shared references of the same thread. Some of the typical access patterns which are likely to be present in HPC applications include:

- *Local shared read/set:* This involves read or write operations on shared memory locations that have affinity to the respective reading or writing thread.
- *Unit stride shared read/set:* Perform read or write operations on consecutive shared memory segments that have affinity to a remote thread.
- *Random shared read/set:* Access a random set of shared memory locations with affinity to a remote thread for reading or writing.
- *Stride-n shared read/set:* Read or write access similar to that of a column-major access in ANSI C. This operation is expected to simulate a non-cache-sensitive (cache unfriendly) access.
- *Unit stride shared copy:* Data is copied sequentially from one memory location to another within the shared memory space. The memory locations are accessed

TABLE 4.1: The NAS Parallel Benchmarks Workload Classes

Benchmark	Size	Operations $(\times 10^3)$	MFLOPS
EP	2^{28}	25.7	209
MG	256^3	3.8	175
CG	$14,000$	1.5	126
FT	$256^2 \times 128$	5.6	189
IS	$2^{23} \times 2^{19}$	0.8	68
LU	64^3	63.5	193
SP	64^3	101.4	214
BT	64^3	179.3	230

(A) Class A workloads Benchmark

Benchmark	Size	Operations $(\times 10^3)$	MFLOPS
EP	2^{30}	100.7	–
MG	256^3	17.9	497
CG	$75,000$	53.8	443
FT	512×256^2	71.4	561
IS	$2^{25} \times 2^{21}$	3.2	242
LU	102^3	321.3	491
SP	102^3	448.7	629
BT	102^3	721.0	573

(B) Class B workloads Benchmark

Benchmark	Size	Operations $(\times 10^3)$	MFLOPS
EP	2^{32}	100.9	–
MG	512^3	26.4	580
CG	$150,000$	70.6	559
FT	512^3	91.3	989
IS	$2^{27} \times 2^{23}$	6.6	304
LU	162^3	400.2	540
SP	162^3	505.5	730
BT	162^3	938.5	609

(C) Class C workloads Benchmark

in a contiguous manner, with no gaps between consecutive elements (unit stride). The shared memory read is followed by a write. Both memory segments involved have affinity to the same remote thread.

- *Random shared copy:* In this case, data is copied between shared memory locations, but unlike unit stride access, the source and/or destination memory locations are accessed in a non-sequential (random) order.

- *Stride-n shared copy:* Data is copied between shared memory segments, and the accessed elements are separated by a fixed interval or "stride" of n positions. The pattern is more general than unit stride (where $n = 1$) and is used in scenarios where data is not contiguous but follows a regular spacing.

- *Local memget:* A contiguous chunk of data is copied from a shared memory location that has affinity to the current thread into a private (local) memory location.

- *Local memput:* Data is written from a local memory region (e.g., a private buffer) to a specific memory location in the shared memory space that has affinity to the writing thread.

- *Local memset:* A specific memory region in the shared memory space, with local affinity to the executing thread, is initialized or filled with a data

- *Local memcpy:* Data is copied between two memory locations, both of which reside in the shared or private memory space associated with the current thread (local affinity).

- *Remote memget:* Data is retrieved from a shared memory location associated with a remote thread and transferred to a local memory region of the calling thread. This is like the *local memget*, but the source has affinity to a remote thread.

- *Remote memput:* Data is written from a local memory region (e.g., a private buffer) to a shared memory location that has affinity to a remote thread.

- *Remote memset:* The operation where a region of shared memory with affinity to a remote thread is initialized or filled with a specified value by the calling thread.

- *On-thread memcpy:* Data is copied between two memory locations within the same thread, either in private memory or in shared memory regions with affinity to the thread. This is like the *local memcpy*, but both the source and the destination are with affinity to the same remote thread.

- *Remote memcpy:* Data is copied between two shared memory locations, with the source and destination residing in memory segments associated with different threads. Like the on-thread memcpy, but the source and the destination have affinities to different remote threads.

4.3 Experimental Results

4.3.1 Experimental setup

The performance of the setup was characterized using the NAS Parallel Benchmarks and the STREAM micro-benchmark. The benchmark suites are executed using up to 128 cores/threads on the 12-node Jaguar cluster. The nodes have a single socket equipped with an 8-core (16 hyper-threaded) Intel Xeon 2.4 GHz processor and 16 gigabytes of DDR3 memory. The nodes are interconnected with a 10Gb Ethernet in a star topology. All codes were implemented in C and compiled with the Berkeley UPC runtime release 2019.4.2 (with the Berkeley UPC-to-C translator version 2.28.0), and MVAPICH2 2.3.2.

4.3.2 NAS Parallel Benchmarks Results

Benchmarking experiments were conducted using multiple kernels from the UPC implementation of the NAS Parallel Benchmarks on the EPGASS setup. Performance was further assessed under different core configurations across the cluster. Figures 4.2–4.6 report the performance measurements for NPB workloads using both UPC and MPI on the setup. Part *A* in each figure shows the timing of the respective workload, whereas, Part *B* shows the corresponding total throughput (Million Operations per second (Mop/s)) for all the test cases. For each benchmark, the experiment was run three (3) times and the average time reported. For MPI implementations, time was reported

using the *MPI_Wtime()*, which provides wall-clock time in seconds with high precision. Similarly, for the UPC implementations, the benchmarks leverage the *upc_ticks_now()* and *UPC_TICKS_PER_SEC* to capture time for each run. The time before the start of the computation is recorded as *t_start*, while the time after the computation is recorded as *t_end*. The total elapsed time is then computed as follows:

$$elapsed_time = t_end - t_start$$

The study identifies substantial performance differences between MPI and UPC implementations across the different benchmark programs. However, unlike most of the other programs, the EP benchmark in figures 4.2a and 4.2b demonstrated a comparable throughput performance between UPC and MPI test cases as the number of processors increase. The EP benchmark is an Embarrassingly Parallel program that requires nearly no effort to communicate data between different tasks, resulting in minimal communication and better throughput. Each processor performs its own tasks (generates random numbers and performs computations on them independently) with minimal need for communication between processors. Both UPC and MPI show good scalability, though UPC performs relatively better (especially for processors: 36, 49 and 128 showing 1.5×, 2.1×, 2.5× improvement respectively) as it handles parallel tasks more efficiently with its partitioned global address space.

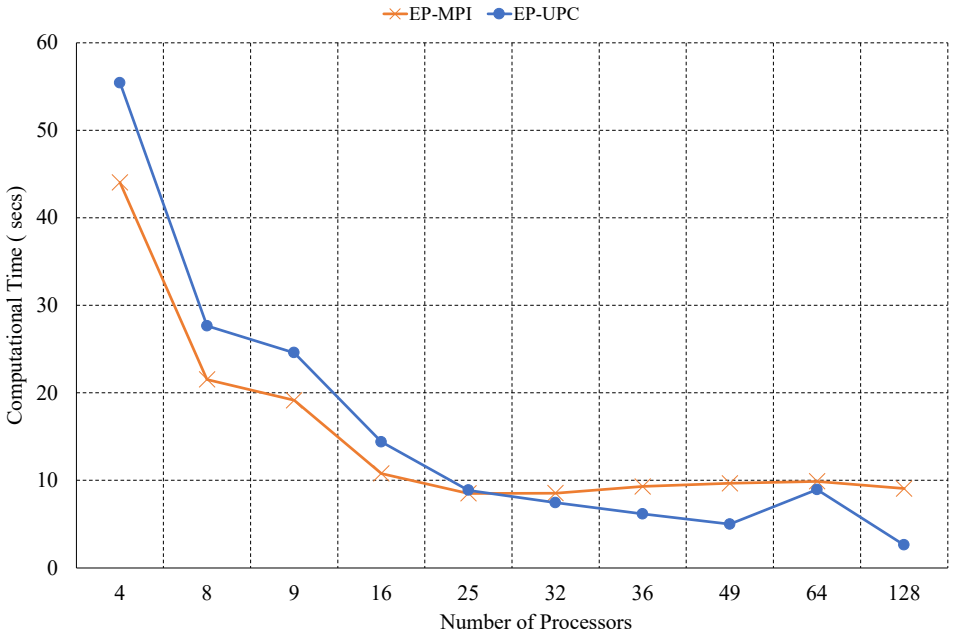
The IS plots in figures 4.3a and 4.3b illustrate that given a small number of processors, both UPC and MPI perform similarly. However, as the number of processors increases, the gap between UPC and MPI widens with UPC maintaining a steady increase in throughput likely due to lower synchronization costs and better memory access patterns. The benchmark performs sorting of large integer arrays, which requires extensive communication to distribute and gather the data. Consequently, this involves random access to memory and significant inter-processor communication. The plots show a steeper decrease in computational time, and corresponding increases in throughput as processor/thread count increases for UPC. UPC's efficient memory access patterns and reduced synchronization costs lead to higher throughput. Unlike UPC, MPI shows a less pronounced increase in throughput, which may be due to synchronization overhead.

The FT (Fourier Transform) benchmark demonstrates a similar pattern as other benchmarks, with MPI performing better than the UPC implementation with lower processor count. The benchmark performs a series of complex Fast Fourier Transforms (FFTs), which require extensive communication and data shuffling involving series of data transpositions and high computational complexity. Figures 4.4a and 4.4b show a general increase in operational throughput and a reduction in time in both implementations. However, unlike UPC, the MPI performance degrades with higher processor count, particularly from 64 cores upward, highlighting higher communication or computation overhead.

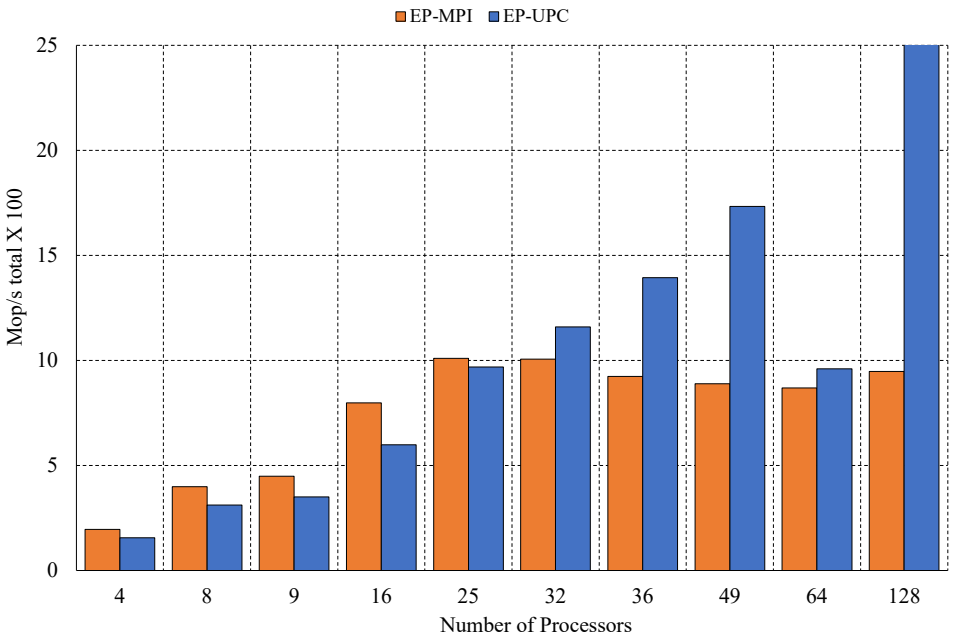
Similarly, the MG benchmark, which solves a 3D discrete Poisson equation using a multigrid method, involves recursive grid computations and significant interprocessor communication. Figures 4.5a and 4.5b illustrate the performance for the MG benchmark, highlighting the performance differences. The plots show that UPC provides relatively better (up to $\approx 10\times$) throughput performance compared to MPI across various processor counts. The MPI implementation shows slower improvements in throughput with an increase in processor counts, pointing to a potential relative increase in synchronization and communication costs.

Regarding the CG benchmark, which involves matrix-vector multiplications, the study observes that computational time decreases with more processors for both models, but UPC shows a more significant reduction in time. Figures 4.6a and 4.6b show that the UPC achieves operational throughput higher than that of MPI in various processor counts. However, the performance gap widens with the number of processors. The throughput of the UPC increases more steadily, reflecting efficient utilization of additional processors and global address space storage. MPI's performance shows a slower rate of increase, indicating potential bottlenecks in communication or computation. The benchmark requires significant communication between processors, which is heavy on both computation and communication, involving iterative solvers.

Figure 4.7 presents the percentage of time used for communication during the IS benchmark. The figure highlights the fact that the interprocess communication increases with increasing number of cores, and a substantial amount of time (up to over 80%) is used



(A) Computation Time

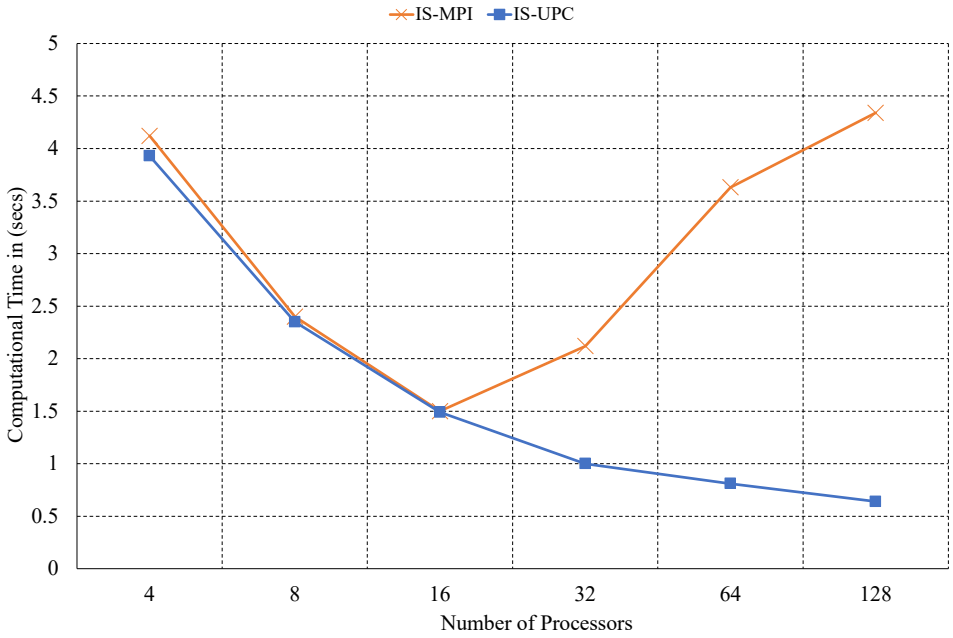


(B) Operational Throughput

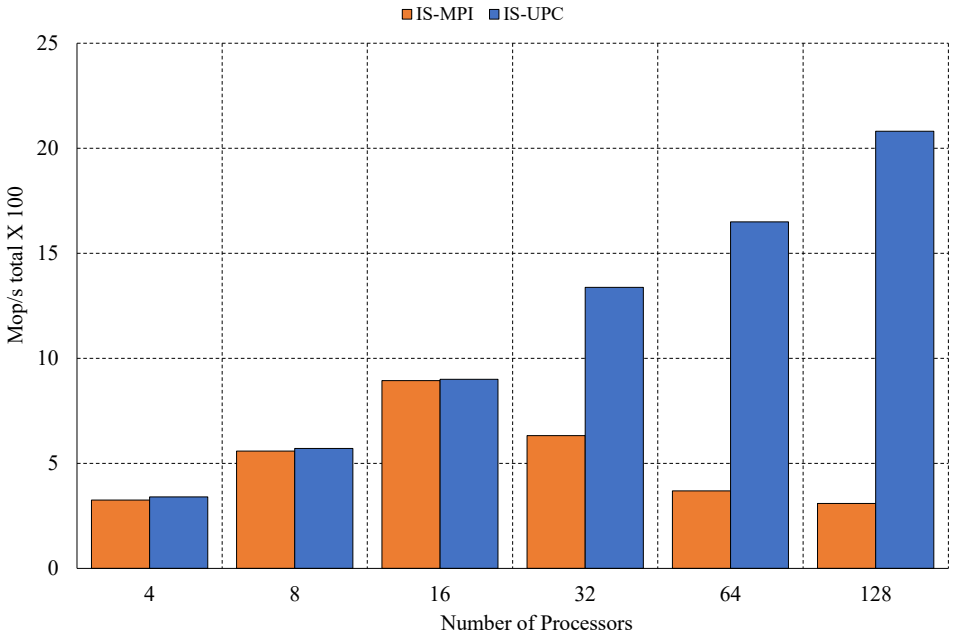
FIGURE 4.2: Evaluation of system performance using the **EP** benchmark comparing UPC and MPI implementations

for communication and synchronisation during the execution of the parallel task. Consequently, impacting on the throughput. The plot shows that up to 85% and 89% (for UPC and MPI, respectively) are used for communication during the integer sort computation across the cluster nodes.

Figure 4.8a illustrates the differences in the amount of memory used by various UPC



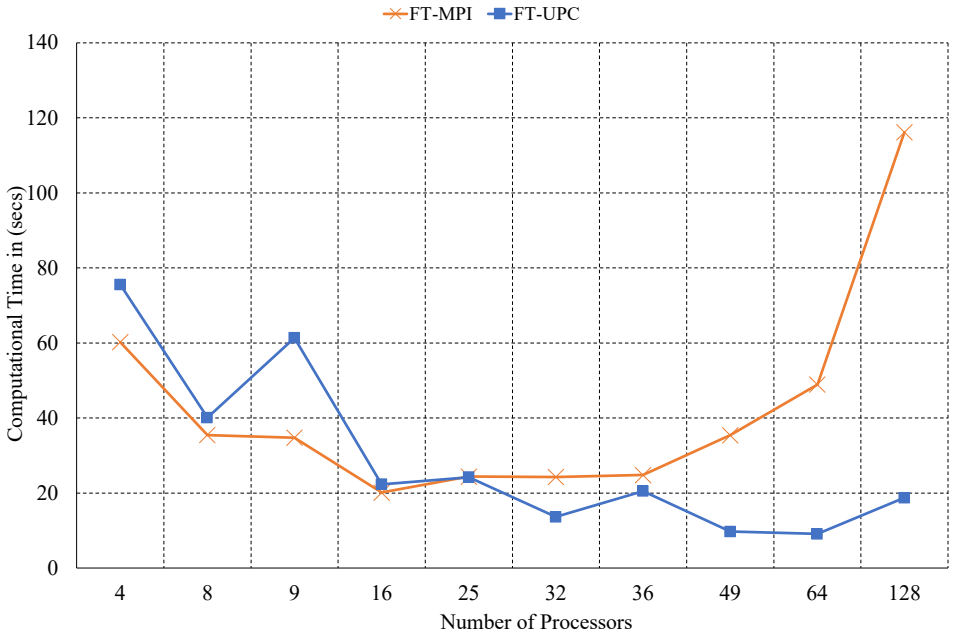
(A) Computation Time



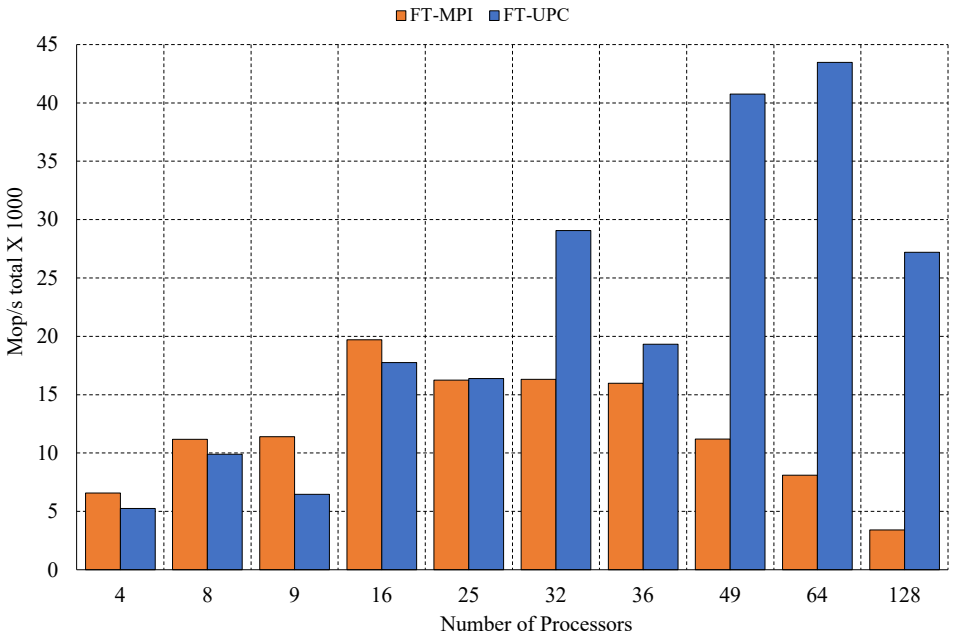
(B) Operational Throughput

FIGURE 4.3: Evaluation of system performance using the **IS** benchmark comparing UPC and MPI implementations

versions of the NPB applications, compared to MPI using the C-class data sets on 128 cores/threads of the setup. The UPC version used relatively a little more memory than the MPI. UPC provides a distributed global shared address space for shared data, with each of the nodes contributing a quota of its memory to the shared storage. A portion of each node's memory is marked as part of the global address space, which could



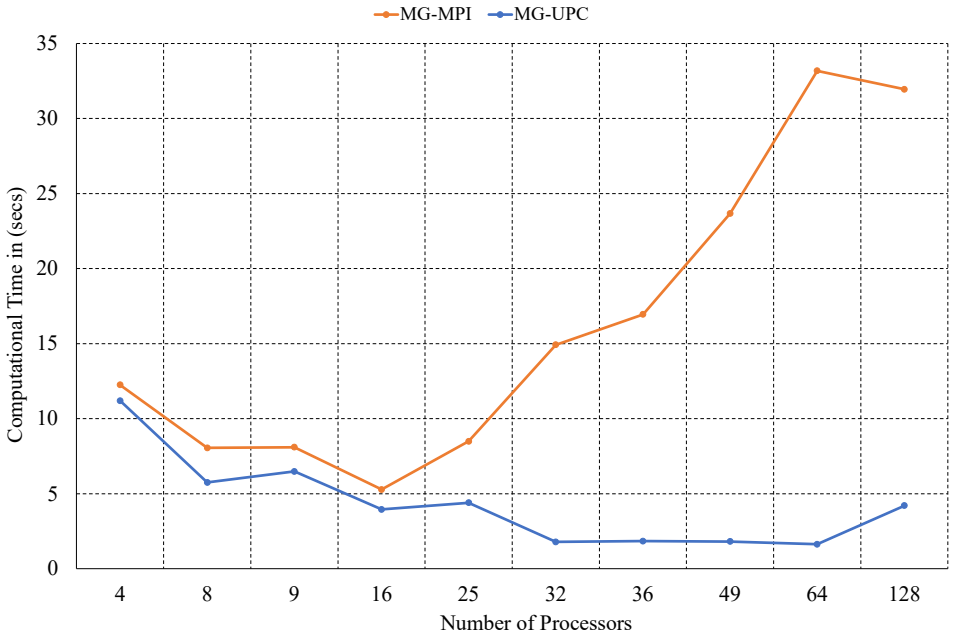
(A) Computation Time



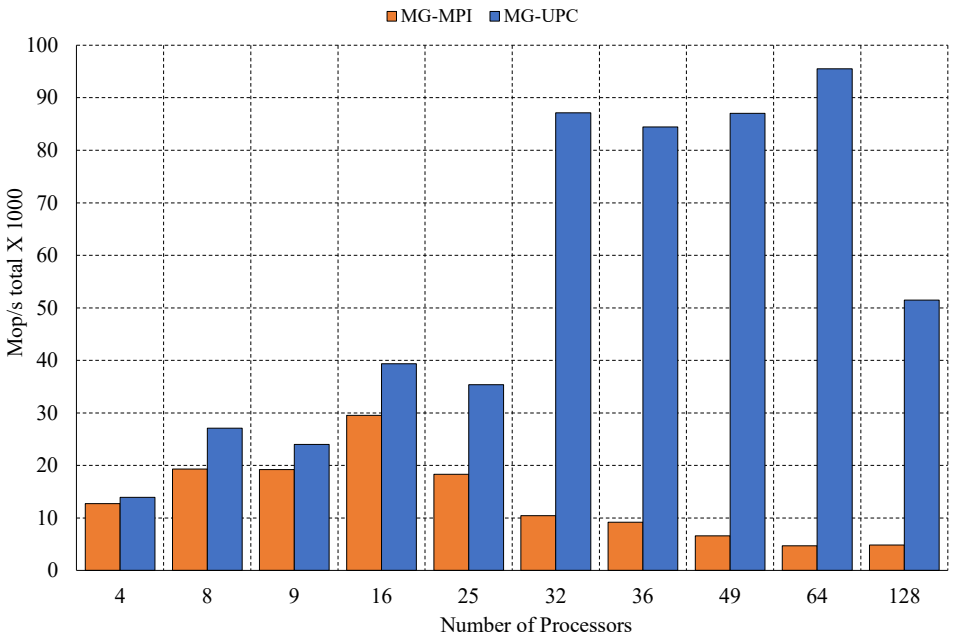
(B) Operational Throughput

FIGURE 4.4: Evaluation of system performance using the **FT** benchmark comparing UPC and MPI implementations

potentially account for the observed memory usage as nodes contribute to the global address space. The results show up to about 12% more memory usage for the UPC compared to the MPI implementation, particularly for the EP benchmark kernel. Figure 4.8b illustrates the actual amount of memory used by each of the kernels. The figure shows a relatively low memory footprint ($\approx$ 1.0% for MPI and $\approx$ 0.9% for UPC)



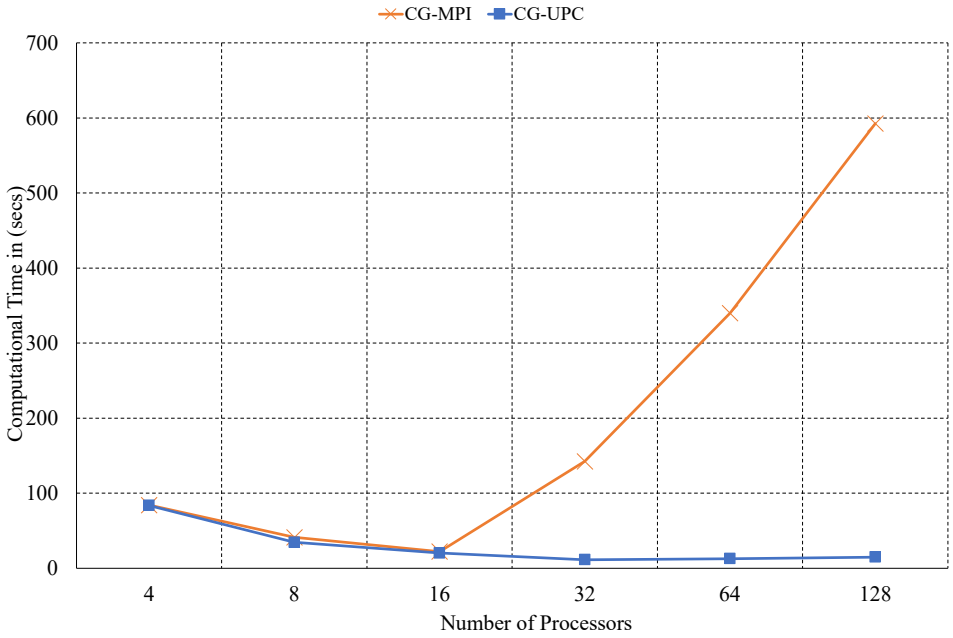
(A) Computation Time



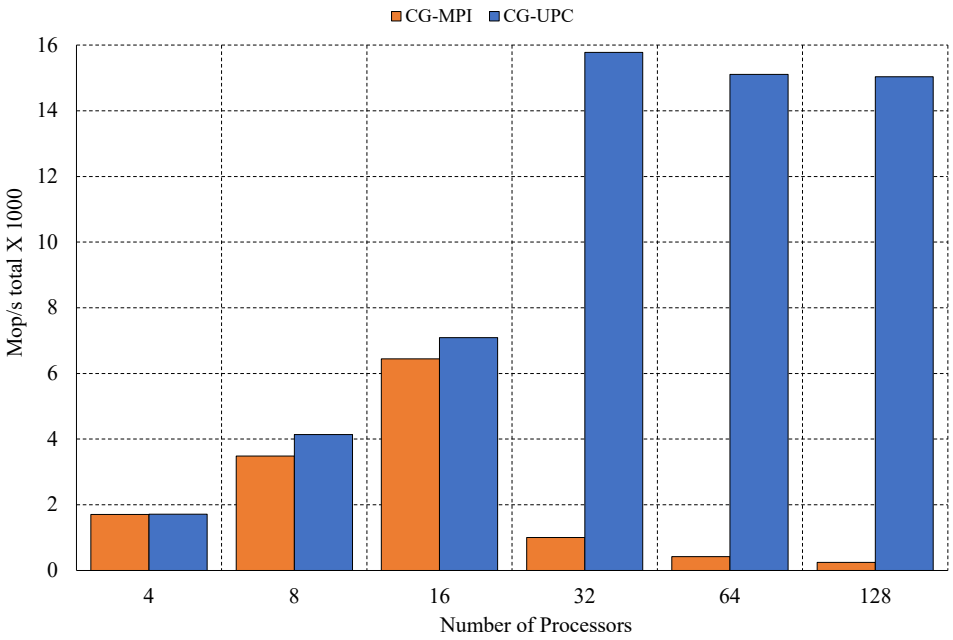
(B) Operational Throughput

FIGURE 4.5: Evaluation of system performance using the **MG** benchmark comparing UPC and MPI implementations

for the IS kernel compared to the EP, FT, CG, and the MG kernels. Both MPI and UPC show similar utilisation patterns, with MPI demonstrating lower utilisations in many instances. The memory footprint is largest for the FT kernel. The FT kernel executes a Fourier transform that incorporates a transpose operation. The implementation of the transpose operation requires an additional array and data buffer, which



(A) Computation Time



(B) Operational Throughput

FIGURE 4.6: Evaluation of system performance using the **CG** benchmark comparing UPC and MPI implementations

results in a large memory footprint. For the CLASS C workload, the workload size is $512 \times 512 \times 512 \times sizeof(\text{double complex}) = 2GB$

As the number of tasks increases, additional memory is primarily required. This is because the runtime system needs to maintain more metadata and control information for managing these tasks, including task identifiers, communication schedules, and state

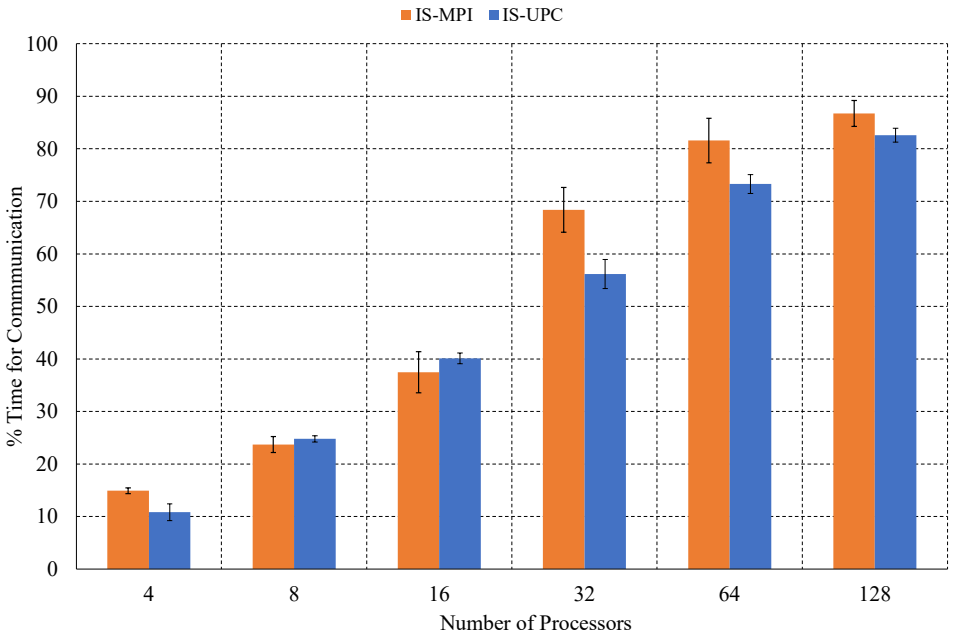
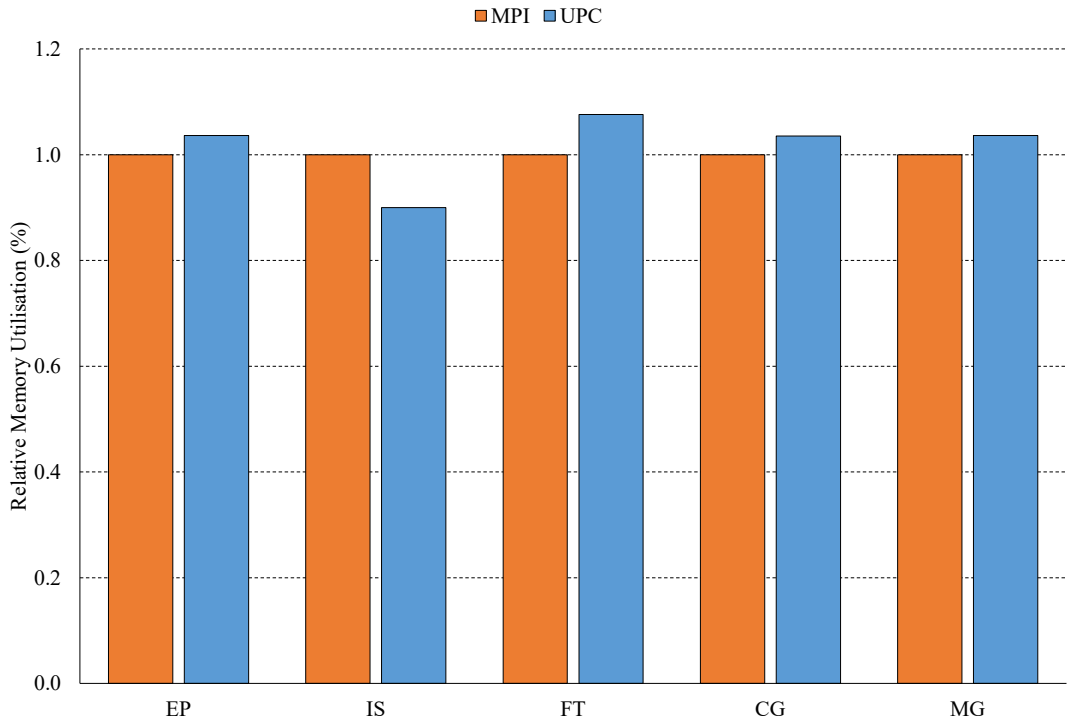
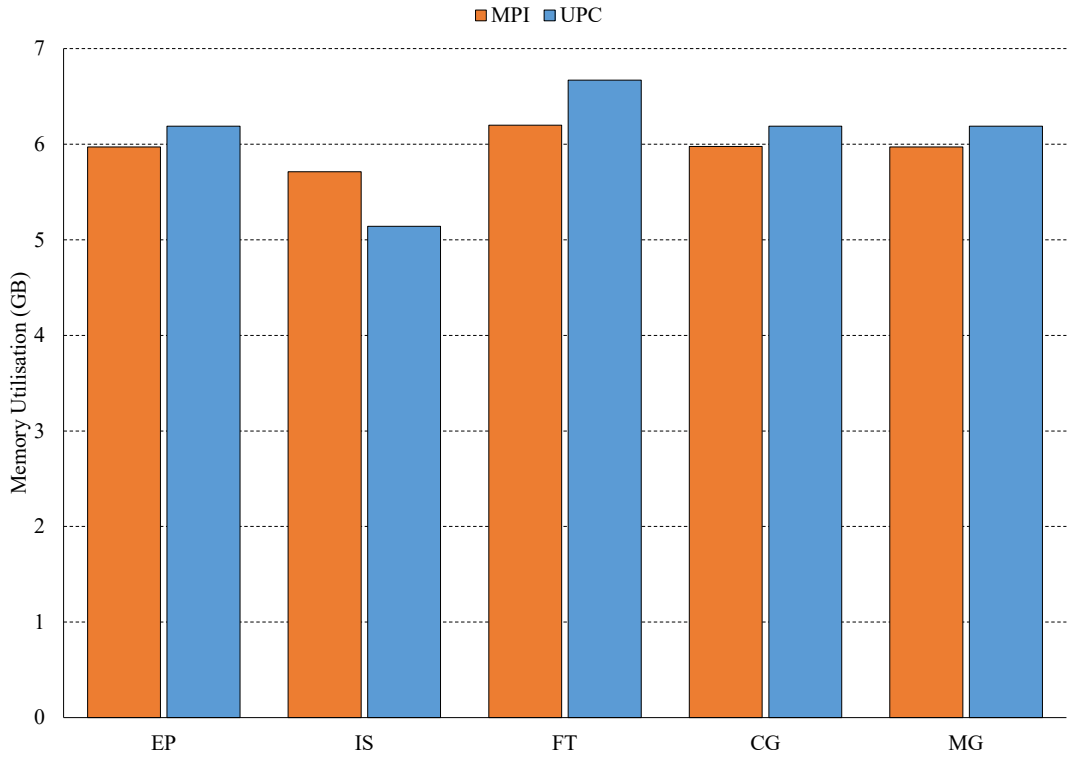


FIGURE 4.7: Proportion of total execution time attributed to communication in the **IS** benchmark showing increasing % of time for communication with increase in number of processors.

information, all of which contribute to increased memory usage. Additionally, each task requires communication buffers to send and receive messages. As the number of tasks increases, the number of possible communication paths grows as well, and hence more buffers are needed to handle these interactions. The size of these buffers can vary depending on the application's communication patterns and the volume of data being exchanged. To minimize communication overhead and latency, MPI applications often replicate data across multiple tasks. This replication ensures that each task has quick access to the necessary data without needing frequent communication with other tasks. However, this replication also increases memory usage, as the same data may be stored multiple times across different tasks. Similarly, for UPC, the memory requirements increase as the number of threads (tasks) increases from the runtime overhead's requirement for maintaining metadata thread states, and synchronization primitives, including shared memory management. Although UPC provides a global address space that simplifies access, the underlying implementation still relies on efficient communication mechanisms that utilize these communication buffers.



(A) Relative memory usage



(B) Memory Usage in GB

FIGURE 4.8: Memory utilisation. (A) shows the relative memory usage of the UPC benchmark applications compared to MPI, whiles (B) illustrates the actual usage in gigabytes

4.3.3 STREAM micro-benchmark Results

The study gathered several measurements using the UPC STREAM microbenchmark on various access patterns that mimic access patterns for real-world scientific applications.

The performance was measured using different access patterns:

- **Local shared accesses** : This involves a local write to or read from shared memory, with the memory having affinity to the thread performing the access. The cost of such access should ideally match that of private memory access. However, system runtime overhead impacts accesses, resulting in performance slightly worse than that of private memory accesses.

- **Remote shared accesses** : Remote shared access refers to a read or write operation in shared memory where the calling thread has no affinity to the memory access region.

- **Block memory operations** : Refers to operations that involve contiguous chunks of memory, typically in a coarse-grained approach. Coarse-grained accesses to shared memory locations utilize UPC string handling functions, involving various combinations of source and destination affinities. These operations are designed to handle larger memory regions in a single call, improving efficiency by reducing the overhead associated with multiple fine-grained memory accesses.

Table 4.2 highlights the performance for the STREAM benchmarks showing results for strided access test across varying threads/processor count. The strided access involves accessing memory locations at fixed intervals (strides). For instance, if the stride is 8, the program accesses every 8th element in the array ensuring access predictability which allows for better prefetching and caching strategies. The table shows an intial high read rates which significantly drop beyond 16 processors suggesting that the memory bandwidth utilization decreases as the number of processors and worker nodes increase, likely due to increased contention overheads. As more processors access the shared memory system, the memory resource contention increases. When multiple processors attempt to read from or write to memory simultaneously, it could create bottlenecks,

TABLE 4.2: UPC STREAM Benchmark highlighting memory bandwidth for Strided access for Reads & Writes

	Array	Mem.	Rates ($\times 10^6$)		Read Time (ms)			Set Time (ms)		
Procs.	($\times 10^5$)	(MB)	Read	Set	Avg.	Min.	Max.	Avg.	Min.	Max.
4	8.0	18.3	999.83	696.73	0.2	0.2	0.2	0.3	0.3	0.3
8	16.0	36.6	995.09	696.73	0.2	0.2	0.2	0.3	0.3	0.3
16	32.0	73.2	1001.03	697.31	0.2	0.2	0.2	0.3	0.3	0.3
32	64.0	146.5	490.28	345.35	0.4	0.4	0.4	0.6	0.6	0.6
64	128.0	293.0	489.13	346.06	0.4	0.4	0.4	0.6	0.6	0.6
128	256.0	585.9	328.45	231.99	0.6	0.6	0.6	0.9	0.9	0.9

reducing the effective memory bandwidth available to each processor. Strided access patterns typically exhibit good locality and predictability, leading to high initial memory bandwidth utilization. However, as the number of processors increases, contention and overhead cost begin to diminish the benefits of additional parallelism, leading to a decrease in memory bandwidth utilization. The *set* rates follow a similar trend as the *read* rates, with a noticeable drop in performance as the processor count increases. Maintaining cache coherence among multiple processors incurs overhead, especially with strided patterns that might cause frequent cache line invalidations and updates. However, a minimal variation is observed across all processor count *read* times, indicating consistent performance for strided reads. similarly, the times for the *set* operation remain fairly consistent, but show a slight increase with higher processor count. Both *read* and *set* operations scale up well to 16 processors. Beyond that, the memory bandwidth utilization begins to decrease, suggesting diminishing returns on performance due to contention overheads.

Unlike strided accesses, random accesses are inherently challenging due to poor locality and an increased likelihood of cache misses. The inherent challenges are further exacerbated by increased contention and overhead with more processors, leading to significant decreases in memory bandwidth utilization. Table 4.3 illustrates the results of the random-access test that show that performance drops more dramatically with

TABLE 4.3: UPC STREAM Benchmark highlighting memory bandwidth for Random Reads & Writes

	Array	Mem.	Rates ($\times 10^6$)		Read Time (ms)			Set Time (ms)		
Procs.	($\times 10^5$)	(MB)	Read	Set	Avg.	Min.	Max.	Avg.	Min.	Max.
4	8.0	18.3	393.65	66.38	0.5	0.5	0.5	3.0	3.0	3.0
8	16.0	36.6	395.13	66.45	0.5	0.5	0.5	3.0	3.0	3.0
16	32.0	73.2	392.91	66.31	0.5	0.5	0.5	3.0	3.0	3.0
32	64.0	146.5	188.51	62.97	1.1	1.1	1.1	6.0	3.2	6.4
64	128.0	293.0	186.41	31.35	1.1	1.1	1.1	6.4	6.4	6.4
128	256.0	585.9	125.86	21.22	1.6	1.6	1.6	9.4	9.4	9.4

an increase in processor count. This is potentially due to high contention, poor locality, and increased synchronization overhead. Additionally, read and set rates are much lower compared to strided access, reflecting the challenges of random-access patterns.

The results from the strided and random-access tests in the UPC STREAM benchmark highlight the inherent challenges and overheads associated with different memory access patterns. The observed decrease in memory bandwidth utilization with increasing processors is due to the compounded effects of contention, cache coherence, synchronization, and inefficient memory access patterns. By optimizing these aspects, performance can be improved, especially for high processor counts.

4.4 Chapter Summary

The study has examined the performance of various benchmarks in the EPGASS setup, comparing the implementations of the UPC and MPI benchmarks for performance of throughput and memory efficiency. Although the overall performance of the NAS Parallel Benchmarks with UPC and MPI are relatively similar, the presented results show some major performance differences. The UPC implementations performed relatively better compared to MPI implementations in most cases including the EP, IS, MG, FT, and CG benchmarks. Overall, the findings indicate that UPC provides significant throughput gains compared to MPI, with performance improvements of up to 2.5×

(EP), 6.5× (IS), 9.5× (FT), 10.0× (MG), and 14.0× (CG). This could be mainly due to the reduced communication costs with the distributed shared-memory model approach.

Across all NPB benchmarks (EP, CG, FT, IS, MG), UPC consistently shows relatively higher throughput compared to MPI, suggesting that the partitioned global address space allows for more efficient computation on the setup compared to MPI. This performance could be attributed to the efficient management of shared memory, and subsequently less communication overhead resulting from efficient data distribution, as well as effective synchronization mechanisms. The partitioned global address space model allows for more direct and efficient access to data, which is crucial in parallel computations. For embarrassingly parallel tasks such as the EP benchmark, UPC's minimal overhead results in significant performance gains. Regarding communication-intensive tasks (CG, FT, IS, MG), efficient data handling and reduced communication latency provide substantial performance gains. The Global Address Space Networking (GASNet) library, being the high-performance communication layer in the UPC runtime, plays a crucial role in facilitating the performance and scalability observed in the UPC benchmarks, including Remote Memory Access (RMA) and Active Messages (AM). It provides a high-performance, as well as scalable communication layer which enables UPC to handle data movement, synchronization, and communication-intensive tasks efficiently. GASNet optimizes the communication layer in UPC, providing low-latency, high-bandwidth communication. This is particularly useful for benchmarks that require frequent communication, such as the CG and FT benchmarks. In particular, it reduces the communication overhead by implementing efficient point-to-point and collective communication operations. This allows UPC to perform better in benchmarks with intensive communication patterns, like MG (Multi-grid) and IS (Integer Sort).

In addition, the study investigated the memory usage of each of the different models. The memory requirements across the different tests were relatively similar ($\approx$ 0.9 *to* 1.1%), although MPI required less memory than UPC in various instances. The UPC runtime maintains additional metadata and control information to manage the increasing number of threads, which could contribute to the observed increased memory usage.

Furthermore, the study evaluated the memory utilisation while using the STREAM benchmark for both strided and random accesses. The results indicated that the memory utilisation relatively decreases with increasing threads for both access methods. However, strided accesses performed much better as expected, pointing to the need for efficient data organisation, and optimizations for distributed parallel processing. The results highlight the inherent challenges and overheads associated with different memory access patterns. Decreasing memory bandwidth utilization with increasing processors is primarily due to increased contention and overhead in accessing shared resources. By understanding these factors, parallel applications can be optimized to mitigate performance degradation and better utilize available memory bandwidth using efficient data partitioning, and algorithms that require less synchronization to exploit more local computation. This optimization is crucial to achieve efficient and scalable parallel computing.

In the next two chapters, the thesis investigates some of these optimisation techniques with optimal data partitioning and layout, leveraging data locality while minimising data movements to enhance overall algorithmic efficiency for in-memory data processing. In Chapter 5, the study explores efficient data partitioning for an augmented matrix-matrix multiplication algorithm (a critical algorithm in linear algebra), while Chapter 6 explores techniques for efficient multidimensional tree index construction and subsequent index queries/searches.

Chapter 5

Matrix-Matrix Multiplication

5.1 Introduction

Distributed computing leverages multiple computational resources to address problems that are too complex for a single system to handle. For computationally intensive tasks, exploiting parallelism is essential to achieving solutions within a reasonable timeframe. Matrix-matrix and matrix-vector multiplications are fundamental linear algebra operations widely used in scientific and numerical computations. When dealing with very large matrices, these operations become highly computationally demanding and require significant memory and processing power.

In the last few decades, many algorithms have been proposed to compute the solution matrix, **C**, by multiplying the matrices **A** and **B**, especially, on distributed parallel systems. The matrix-matrix multiplication equation in distributed parallel systems is typically given as:

$$\mathbf{C} \leftarrow \alpha(\mathbf{A} \times \mathbf{B}) + \beta C$$

where: $\alpha, \beta \in \mathbb{R}$, are some constants and $C \in \mathbb{R}^{m \times p}$, is the resulting $m \times p$ matrix (also used as input for accumulation).

Some of the well-known algorithms include the Shared and Remote-memory based Universal Matrix Multiplication Algorithm (SRUMMA) [98], the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [99], the Parallel Universal Matrix Multiplication Algorithm (PUMMA) [100], and Cannon's algorithm [101]. Various studies

have explored distributed algorithms for matrix-matrix multiplication, particularly for distributed parallel systems [102–110]. Most of these algorithms are highly efficient when implemented using a grid of processors where the matrices **A** and **B**, have been blocked into vertical and horizontal chunks, and distributed across the processors. A key advantage of this approach is that smaller blocks can effectively utilize data locality on individual processors. These sub-blocks are loaded into the fast local memory of the compute nodes, allowing their elements to be re-used multiple times. Typically, these algorithms are implemented using one- or two-dimensional block-cyclic partitioning. This type of data partitioning enhances performance significantly when data re-use is maximised, and data movement is minimised.

In this chapter, the study presents the *Single Matrix Block Shift (SMBS) algorithm*, a scalable parallel algorithm for dense matrix multiplication, specifically optimized for a 2D grid of processors. The algorithm is a variation of Cannon's matrix algorithm, designed to improve performance by limiting block shifting to a single matrix. This approach enhances data locality and minimizes data movement, thereby optimizing computational efficiency. The study provides analytical and experimental comparative analyses of the SMBS algorithm with the Cannon's algorithm and variants of the SRUMMA algorithm on a 2D grid of processors.

The primary contribution presented in this chapter is the implementation of the SMBS dense matrix-matrix multiplication algorithm for distributed-memory systems, which exploits data locality, and re-use to maximize performance. The main contributions reported include:

i. the implementation of a blocked dense parallel algorithm for matrix-matrix multiplication which is optimised for distributed parallel systems, leveraging in-memory data processing and data locality for improved algorithmic efficiency.

ii. the analytical and experimental evaluations employed to assess the efficiency of the SMBS multiplication algorithm compared to the Cannon and the SRUMMA matrix multiplication functions.

5.2 Background

Over 30 years ago, Gupta et al. [111] demonstrated that the performance of parallel matrix-matrix multiplication algorithms vary under different conditions. Since then, several techniques for parallelising matrix-matrix multiplication algorithms on distributed systems have been widely researched [102–105]. Modern computer architectures incorporate hierarchical memory systems, where accessing data from higher levels in the hierarchy offers faster performance compared to lower levels. To fully leverage these systems, efficient algorithms are developed to maximize utilization of the top levels of the memory hierarchy, thereby reducing costly accesses to the lower levels. In dense linear algebra operations, for example, the use of block-partitioned algorithms has been widely studied. This approach reformulates algorithms to operate on sub-matrices, enabling better exploitation of the memory hierarchy and improved performance. Variants of this technique have been extensively applied in matrix-matrix multiplication routines for distributed-memory systems.

A notable example of such an algorithm is the *Shared and Remote-memory based Universal Matrix Multiplication Algorithm (SRUMMA)* [98], designed for both distributed-memory and shared-memory parallel systems. For the computation of $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, SRUMMA assigns each process the relevant blocks from matrices $\mathbf{A}$ and $\mathbf{B}$, performs their multiplication, and stores the result in a locally managed portions of $\mathbf{C}$. Processors access non-local blocks depending on whether these blocks are located in other shared memory segments or within the same memory space [98]. SRUMMA draws on the underlying *Remote Memory Access (RMA)* to autonomously access matrix blocks, eliminating the need for coordination with the processors that own the blocks. This approach, however, incurs an initial runtime cost associated with sorting a task list (including recording) to evaluate and optimize data locality references. [112].

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) is another widely recognized matrix multiplication routine, currently implemented in the Scalable Linear Algebra Package (ScaLAPACK) [113, 114] and other associated parallel software libraries. The ScaLAPACK library comprises a subset of LAPACK [115, 116] routines

which are optimized for distributed-memory systems with multiple instruction, multiple data (MIMD) architectures. In SUMMA, matrix **A** is decomposed into a number of columns, while matrix **B** is decomposed into rows of fixed-size blocks. The column chunks of matrix **A** multiply the respective row chunks of matrix **B** until the final column chunk of **A** is multiplied with the final row chunk of **B** [115] to produce the product **C**.

Choi [117] introduced the *Distribution-Independent Matrix Multiplication Algorithm (DIMMA)* as an alternative to SUMMA. In contrast to SUMMA, DIMMA utilizes a modified pipelined communication strategy to efficiently overlap communication with computation. This approach leverages the least common multiple (LCM) block concept to optimize block sizes for maximum performance. The core idea of the LCM method is to simultaneously process multiple small columns of **A** blocks and a corresponding number of small rows of matrix **B** blocks. This enables the processors to compute the product of numerous tiny sub-matrices from matrices **A** and **B** simultaneously to yield high performance [117].

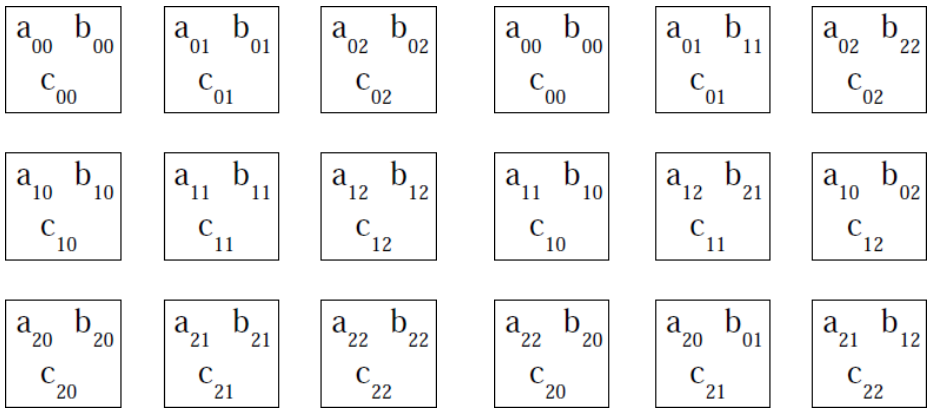
Instead of distributing a single column strip of matrix **A** and a single row strip of matrix **B**, this approach ensures that a column of processors broadcasts multiple columns of **A** along the corresponding processor rows. The distance between these blocks is determined by the least common multiple (LCM) in the column direction. [117].

Huang and Chow [105] developed the *Communication-Avoiding 3D Matrix Multiplication (CA3DMM) algorithm*. The CA3DMM algorithm is designed to reduce data exchange volumes for various input matrix shapes. It focuses on explicitly arranging communication patterns to ensure lower communication costs for distributed-memory parallel general matrix-matrix multiplication (PGEMM). The algorithm further uses a simplified implementation without requiring low-level optimization for performance. Unlike the SUMMA algorithm, CA3DMM starts with an initial computation of an optimal or near-optimal 3D process grid based on the dimensions of the input matrices. Thereafter, the algorithm performs the matrix multiplication using multiple low-rank updates. SUMMA performs different low-rank updates sequentially, whiles CA3DMM performs both calculations in each low-rank update and the computations of different

low-rank updates in parallel.

The Cannon algorithm is a widely used matrix-matrix multiplication method, particularly well-suited for distributed parallel systems. A notable advantage of this algorithm is its memory efficiency. It is based on a $P \times P$ logical or physical grid of processors, employing a blocked data distribution strategy where each processor manages multiple consecutive data blocks. The Cannon algorithm is highly effective for homogeneous 2D grids but poses challenges when adapting it to heterogeneous 2D grids. In Cannon's algorithm, given a 2-dimensional processor grid, the processor P_{ij} $(0 \leq i, j \leq P-1)$, has the sub-blocks in position ij $(0 \leq i, j \leq P-1)$ of matrices A, B and C. From this data distribution, the matrices **A** and **B** blocks are skewed or reassigned such that given the 2D grid of **P** $\times$ **P** processors, the elements of sub-matrix **A** in row i and column $(j+i)$ mod P, i.e., $a_{i,(j+i) \mod P}$, and also the element or sub-matrix of **B** in row $(i+j) \mod P$ and column j, i.e., $b_{(i+j) \mod P,i}$, are allocated to the processor P_{ij}. Particularly, each data of row i, for $(0 \leq i \leq P-1)$, of the sub-matrices of **A** is shifted i times towards the left grid of processors, while each column j, for $(0 \leq j \leq P-1)$ of the elements or sub-matrices of **B** is transferred j times towards the upper grid of processors.

The Figure 5.1a presents the initial block allocations, while Figure 5.1b presents the initial relocation (skewing of the **A** and **B** blocks), imposed by Cannon's Algorithm. The initial setup stage incurs additional computational cost, which affects the overall performance of the algorithm.



(A) Initial assignment (B) Initial relocation

FIGURE 5.1: Initial Data Allocation for Cannon's Algorithm on 3×3 Grid

Matrix-matrix multiplication in distributed-memory systems typically requires substantial data movement, posing challenges in reducing synchronization overhead among the nodes. The cost of data transfer between processing units has long been significantly higher than the cost of arithmetic operations. As a result, minimizing communication overhead in distributed-memory parallel algorithms has been extensively studied, particularly in algorithms for distributed-memory parallel dense general matrix-matrix multiplication (PGEMM). Thus, the efficiency of these distributed algorithms largely relies on the effective distribution of data and the parallel organization of the processing nodes to minimize data transfers. It is therefore unsurprising that state-of-the-art linear algebra libraries, such as LaPACK [118] and ScaLAPACK [119], encounter performance degradation on certain multi-socket systems with multicore processors, such as Symmetric Multi-Processor (SMP) architectures. This is mainly due to the fact that some of the algorithms are not able to completely leverage thread-level parallelism and data locality. Leveraging data locality becomes feasible when data is re-used multiple times, thereby increasing cache hits and improving overall algorithmic performance.

The next section (Section 5.3) discusses in detail the proposed algorithm, which builds on the Cannon matrix-matrix multiplication algorithm. However, unlike the Cannon's algorithm, this study's approach eliminates the initial block relocation or skewing of the **A** and **B** matrices during the initialisation stage. Additionally, the logical shift of the blocks of matrices **A** and **B** onto neighbouring processors in the *2D* grid is limited to only a single matrix. Consequently, the approach promotes data re-use and exploits data locality to improve the computational efficiency. Furthermore, the study enhances the efficiency of the algorithm by overlapping communication with computations. The removal of the sender-receiver synchronisation for the matrices **A** or **B** (as in Cannon's algorithm) significantly enhances overall algorithmic efficiency.

5.3 Single Matrix-Block-Shift Algorithm

5.3.1 Blocking SMBS Algorithm

Given the matrices

$$\mathbf{A}_{M,K} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,K} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,K} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,K} \\ \vdots & \vdots & \ddots & \vdots & \\ a_{M,1} & a_{M,2} & a_{M,3} & \cdots & a_{M,K} \end{pmatrix} \text{ and } \mathbf{B}_{K,N} = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,N} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,N} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,N} \\ \vdots & \vdots & \ddots & \vdots & \\ b_{K,1} & b_{K,2} & b_{K,3} & \cdots & b_{K,N} \end{pmatrix}$$

consider the matrix-matrix product

$$\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}, \text{where } c_{i,j} = \sum_{k=1}^{K} a_{i,k} b_{k,j}$$

and **A**, **B** and **C** are dense matrices.

In essence, the $c_{i,j}$ of matrix **C** can be computed in parallel by taking the dot product of each row i of matrix **A** and column j of matrix **B** on a processor. Unfortunately, this technique does not scale well since as many as MN processing cores are expected to calculate all sub-products, $c_{i,j} \in \mathbf{C}$.

Given that there are $m \times n$ 2D processor grids, the block sizes of the **A**, **B** and **C** matrices are $\frac{M}{m} \times \frac{K}{k}$, $\frac{K}{k} \times \frac{N}{n}$ and $\frac{M}{m} \times \frac{N}{n}$ respectively. The process $P_{i,j}$ computes all $c_{i,j}$ in block $\mathbf{C}^{+}_{i,j}$, with $\mathbf{C}^{+}$ denoting a block in the **C** matrix. Likewise, the blocks in matrix **A** are defined as $\mathbf{A}^{+}$, and those in **B** are defined as $\mathbf{B}^{+}$. The setup of the study's approach is similar to that of the Cannon's algorithm, but with some major differences. In contrast to the Cannon's algorithm, this study divides the **A** matrix into column-wise blocks and that of **B** into row-wise blocks or vice versa. In addition, the proposed algorithm is not restricted to a square grid of processors. For the purposes of this presentation, this discussion is limited to the case where matrix **A** is blocked column-wise and matrix **B**

row-wise. Given the dense matrices **A** and **B** blocked into 4×4 matrices as illustrated in Figure 5.2, the implementation of the proposed algorithm on a grid of distributed parallel processors is as follows:

1. Partition matrices **A** and **B** into $m \times n$ blocks, and distribute **A** column-wise and **B** row-wise so that $A^+_{0,0}$ and $B^+_{0,0}$ are located on processor $P_{0,0}$, $A^+_{0,1}$ and $B^+_{1,0}$ on processor $P_{0,1}$, etc. The blocks are distributed such that the column count of matrix **A** is the same as the row count of matrix **B**. Figure 5.3 highlights the data distribution strategy.

2. Create a row communicator for each row of the 2D grid. This is useful to aggregate the final $C^+_{i,j}$ per each iteration.

3. The algorithm then enters the main computational loop to compute local block multiplication $m - 1$ times. For a square grid, where $m = n$, the routine executes in $\mathcal{O}(\sqrt{m^2 - 1})$ time.

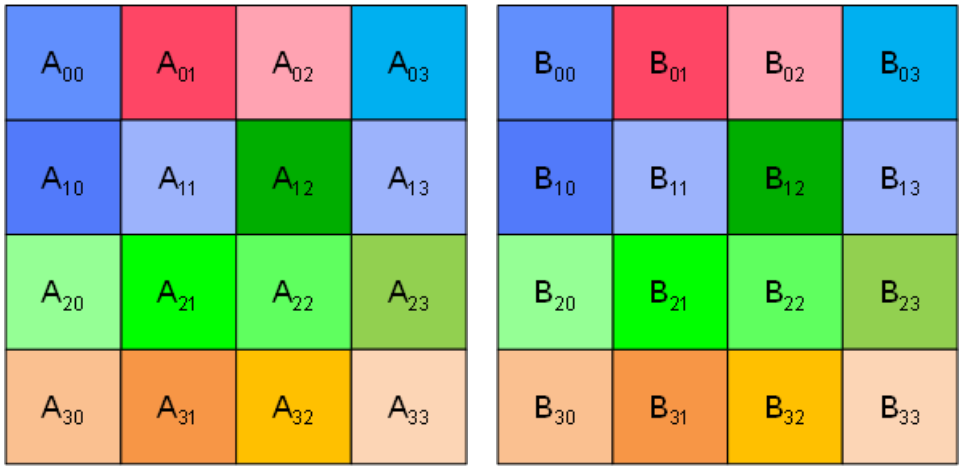
4. For each step in the computational loop, call **Reduce** on the row communicator to get the sum of all the partial sums, i.e., $\sum_{j=0}^{col_size} c_{i,j}$. The final $C^+_{i,j}$ block of **C** results from this output. The rank of the processor $P_{i,j}$ that will hold the $C^+_{i,j}$ is then computed as follows:

$$(i + row) \mod col_size.$$

   where i is the i-th iteration, *row* is the row number, and *col_size* is the total number of columns per row.

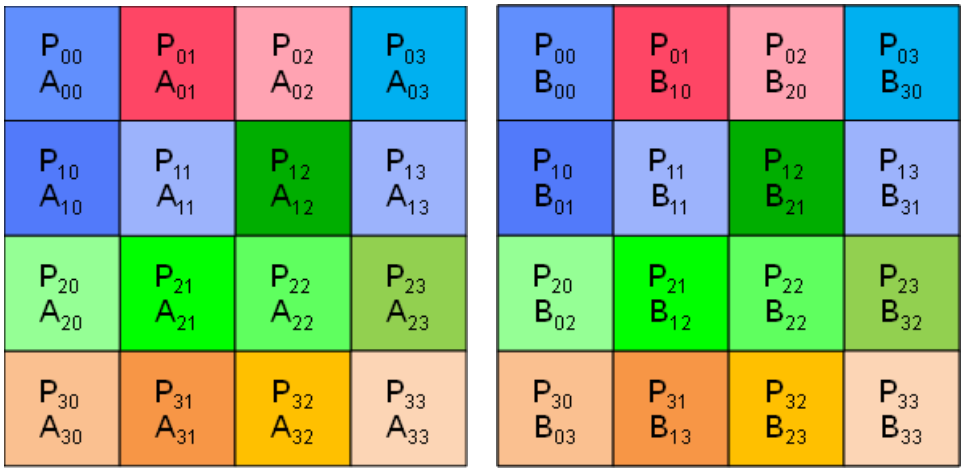
5. Every block of **B** is shifted (moved) one step up with wrap-around. In contrast to the Cannon algorithm, the blocks of matrix **A** are not shifted. This facilitates data re-use and data locality, resulting in improvement in the overall algorithmic performance.

6. Perform the next block multiplication, and then **Reduce** the partial result per row. This is repeated $m - 1$ times.



(A) Matrix A (B) Matrix B

FIGURE 5.2: Block of Matrices A and B



(A) Distribution of **A** (B) Distribution of **B**

FIGURE 5.3: Block Distribution of Matrices A and B

The SMBS routine is formally illustrated in Algorithm 5.1.

Given the dense matrices **A** and **B** which have been partitioned/blocked in into 4×4 sub-matrices as illustrated in Figure 5.2, the sub-matrices are distributed on the 2D 4×4 grid of processors as demonstrated in Figure 5.3. During the initialisation stage, the A and B matrices are set up such that the processor $P_{i,j}$ has blocks $A^{+}_{i,j}$ and $B^{+}_{j,i}$. The product of $A^{+}_{i,j}$ and $B^{+}_{j,i}$ is computed per processor, and a reduction per row results in the final $C_{i,j}$. Figure 5.4d depicts the $C_{i,j}$ blocks computed during each i_{th} iteration. In the 4×4 example, the reduction per row results in the diagonal matrix $C_{i,i}$ on processors $P_{i,i}$ where $i \longleftarrow \{0, 1, 2, 3\}$ during stage 1 of the algorithm. Similarly, the per row reduction in the 2nd iteration results in the $C^{+}_{0,1}, C^{+}_{1,2}, C^{+}_{2,3}, C^{+}_{3,0}$ on the $P_{0,1}, P_{1,2}, P_{2,3}, P_{3,0}$ processors,

Algorithm 5.1: SMBS Algorithm

Data: $A[M][K], B[K][N]$, local blocksize *nlocal*, *gridsize* $m \times n$

Result: $C[M][N]$

begin

Distribute A column-wise and B row-wise across the $m \times n$ grid

for $r \leftarrow 0$ **to** $m - 1$ **do**

MatrixMultiply$(nlocal, A_{i,k}, B_{k,j}, C^{+}_{i,j})$

ReduceMatMultPerRow$(C^{+}_{i,j}, C_{i,j})$

$j \longleftarrow (row + r) \mod p$

$C_{r,j} \longleftarrow \sum_{j=0}^{p-1} C^{+}_{r,j}$

Up-circular-shift each column of B by 1,so that $B_{i,j}$ *is overwritten by* $B_{i,(j+1) \mod p}$

return C

Algorithm 5.2: MatrixMultiply

Data: $n, *a, *b$, local blocksize *nlocal*

Result: $*c$

begin

```
// Matrix multiplication
```

$n \longleftarrow nlocal$

for $i \leftarrow 0$ **to** $n - 1$ **do**

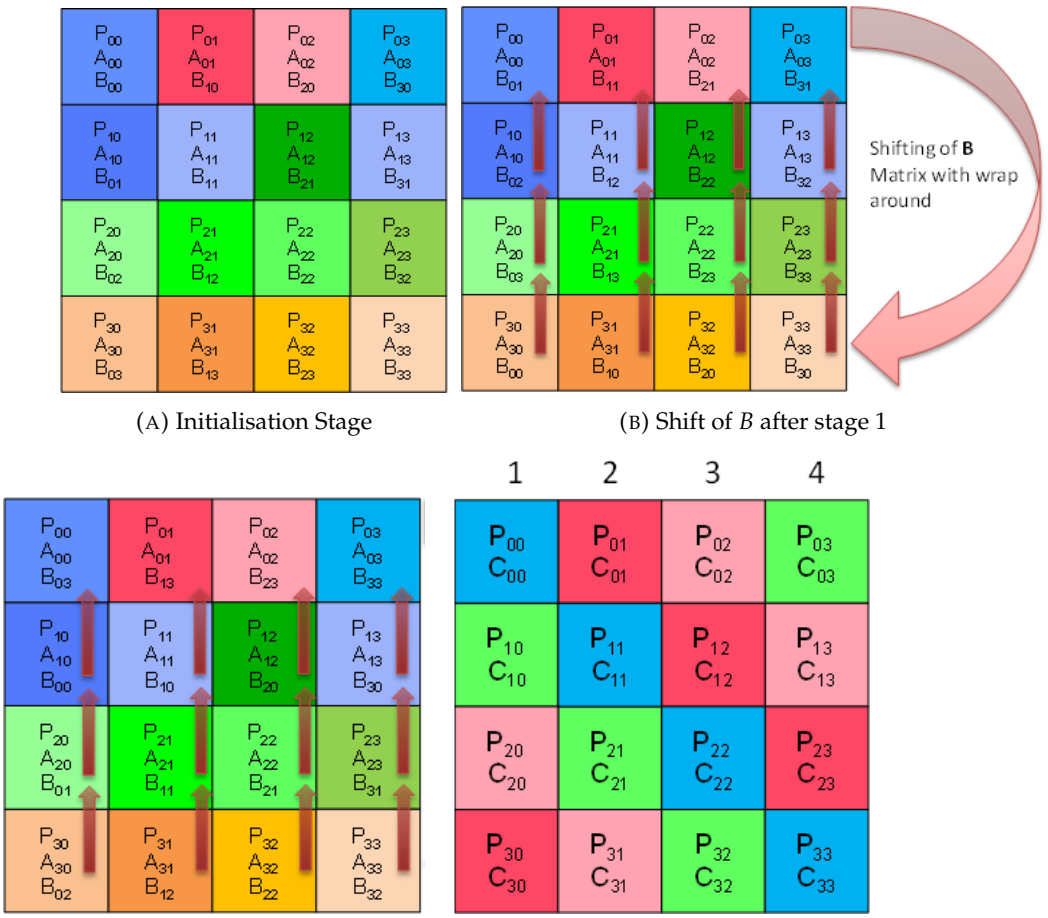
for $j \leftarrow 0$ **to** $n - 1$ **do**

for $k \leftarrow 0$ **to** $n - 1$ **do**

$c[i * n + j] \leftarrow c[i * n + j] + a[i * n + k] \times b[k * n + j]$

return c

respectively. It is important to note that all processor rows execute this reduction simultaneously. This reduction step is overlapped with the shifting of the B matrix blocks, as indicated in Figures 5.4b and 5.4c.



(A) Initialisation Stage

(B) Shift of B after stage 1

(C) Block layout of A & B in stage 4 (*i.e., 3rd iteration*)

(D) Results per iteration after **Reduction** Stage

FIGURE 5.4: Sketch of SMBS Algorithm

It should be noted that each i-th iteration outputs the final $C^{+}_{i,j}$ in the $P_{i,k}$ processor of that row where $k = (i + row) \mod col_size$. Consequently, a total of $C^{+}_{i,j}$ row blocks is produced per iteration. This is particularly significant in certain image processing applications, where specific portions of the final image are needed during the processing stage. The completed blocks can be asynchronously gathered for further processing while subsequent iterations proceed. The overlapping of these operations offers a performance advantage compared to existing algorithms.

5.3.2 Non-Blocking SMBS Algorithm

The SMBS algorithm, limits the amount of data block movements by half compared to the original Cannon routine. The algorithm's performance can be enhanced by incorporating non-blocking message passing, allowing computation and communication to overlap. In large-scale applications involving large matrices, communication often

accounts for more than half of the total latency in a distributed system. Redesigning algorithms to overlap communication with computation can significantly improve application performance.

The study implements non-blocking routines in the main computational loop of the SMBS algorithm, depicted in Algorithm 5.3. This thesis refers to the non-blocking implementation of SMBS as NB_SMBS. In the case of the NB-SMBS, the study leverages the non-blocking point-to-point communication features of the Message Passing Interface (MPI), specifically *Isend* and *Irecv*, to send and receive block buffers of matrix **B** before the call to the *MatrixMultiply* computation. It is worth noting that, unlike Cannon's algorithm, the block shifts in this case are limited to a single matrix. The trade-off in the non-blocking routine is the need for additional buffers to store the incoming blocks. The non-blocking SMBS takes advantage of the RDMA capabilities of the underlying inter-node communication, resulting in substantial performance enhancements. The experimental results presented in later sections suggest over $5\times$ performance gains over blocking SMBS.

5.4 Analysis of the algorithm

It is worth noting that a seemingly simple algorithm, consisting of a single statement and three nested loops, like matrix-matrix multiplication, has attracted substantial research attention over the past few decades. This is due to the fact that matrix-matrix multiplication serves as a fundamental operation in a wide range of scientific applications. Matrix-matrix multiplication generally has $\mathcal{O}(N^3)$ algorithmic complexity, and therefore the numeric applications that depend on them often dedicate significant time to complete the matrix operations. Many research efforts have focused on reducing the $\mathcal{O}(N^3)$ time complexity associated with matrix-matrix multiplication routines. Strassen's algorithm [120, 121] is a groundbreaking and widely used fast method that reduces the computational complexity of matrix-matrix multiplication from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^{2.81})$ operations. Coppersmith and Winograd's algorithm [122] further reduced the complexity to $\mathcal{O}(N^{2.38})$. However, these methods compromise numerical stability, achieving lower complexity through additional operations.

Algorithm 5.3: Non-blocking SMBS

Data: $A[M][K], B[K][N]$, local blocksize *nlocal*, **b_buffers[2]*, gridsize $m \times n$

Result: $C[M][N]$

begin

Distribute A column-wise and B row-wise across the $m \times n$ grid

for $r \leftarrow 0$ **to** $m - 1$ **do**

`// Initiate non-blocking send`

MPI_Isend(b_buffers$[r \mod 2]$, *nlocal* $\times$ *nlocal*, *MPI_FLOAT)*

`// Initiate non-blocking receive`

MPI_Irecv(b_buffers$[(r+1) \mod 2]$, *nlocal* $\times$ *nlocal*, *MPI_FLOAT)*

MatrixMultiply$(nlocal, A_{i,k}, B_{k,j}, C^{+}_{i,j})$

ReduceMatMultPerRow$(C^{+}_{i,j}, C_{i,j})$

$j \longleftarrow (row + r) \mod p$

$C_{r,j} \longleftarrow \sum_{j=0}^{p-1} C^{+}_{r,j}$

Up-circular-shift each column of B by 1, so that $B_{i,j}$ is overwritten by $B_{i,(j+1) \mod p}$

return C

Consider the matrix-matrix product routine $\mathbf{C} \longleftarrow \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $M \times K$, $K \times N$, and $M \times N$ matrices respectively. The following notation is used:

t_s as cost for initialisation

P as number of Processors

$p \times q$ as the size of the 2D processor grid,

i.e. $p \times q = P$

T_{trans} as data transfer (shift) latency

n_i as number of iterations $= \sqrt{P} - 1$,

i.e. the number of row blocks of B,

For simplicity, the analysis assumes that $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ are dense square matrices on a $2D$ processor grid with the matrices distributed as demonstrated in Figure 5.3. Each

processor is assigned a block from the matrices **A**, **B**, and **C** of sizes $\frac{M}{p} \times \frac{K}{q}$, $\frac{K}{q} \times \frac{N}{p}$ and, $\frac{M}{p} \times \frac{N}{q}$ respectively. The total running time of the algorithm, T_{total} is expressed as:

$$
\begin{aligned}
T_{total} &= \text{Time per iteration + startup cost} \\
T_{total} &= T_{row} \times n_i + t_s \\
&\textit{where} \\
T_{row} &= T_{trans} \text{ + compute-time, and} \\
T_{compute} &= \left(\frac{M}{p} \times \frac{K}{q} \times \frac{N}{p}\right) \times t_c \\
\text{Let } M &= K = N, \text{ and let } p = q \text{ then} \\
T_{compute} &= \left(\frac{N}{q} \times \frac{N}{q} \times \frac{N}{q}\right) \times t_c \\
&= \frac{N^3}{q^3} t_c = \frac{N^3}{P^{\frac{3}{2}}} t_c | P = q^2
\end{aligned}
$$

where t_c = cost factor for computing.

T_{trans} could be comparatively much smaller as data is shifted in blocks concurrently. The non-blocking implementation of SMBS overlaps the data transfer with computation and thus, $T_{trans} \approx 0$.

$$
\begin{aligned}
T_{trans} &\approx 0 \\
T_{row} &= T_{trans} + \frac{N^3}{P^{\frac{3}{2}}} \times t_c \\
T_{total} &= \left(T_{trans} + \frac{N^3}{P^{\frac{3}{2}}} \times t_c\right) \times \left(\sqrt{P} - 1\right) + t_s \\
&= \left(\frac{N^3}{P^3}\right) t_c - \left(\frac{N^3}{P^{\frac{3}{2}}}\right) t_c + T_{trans}\sqrt{P} - T_{trans} + t_s
\end{aligned} \tag{5.1}
$$

assuming a network with enough bandwidth capacity,

$$
\begin{aligned}
ts &\approx 0 \\
T_{total} &= \mathcal{O}\left(\frac{N^3}{P^3}\right) - \mathcal{O}\left(\frac{N^3}{P^{\frac{3}{2}}}\right)
\end{aligned} \tag{5.2}
$$

Eqn 5.2 demonstrates that SMBS has theoretical algorithmic complexity which is comparable to well-known algorithms such as SRUMMA, and the Cannon algorithm. But,

unlike SRUMMA which has theoretical algorithmic complexity bounded b, $\mathcal{O}\left(\frac{N^3}{P}\right) + \mathcal{O}\left(\frac{N^2}{\sqrt{P}}\right) + \mathcal{O}\left(\sqrt{P}\right)$, the analysis indicates a further reduction in the theoretical time complexity for SMBS, particularly for some increasing workloads. This translates into potential superior operational throughput and lower latencies.

5.5 EPGASS and the Single Matrix-Block-Shift Algorithm

5.5.1 The General UPC algorithm

The PGAS implementation of SMBS enhances data locality, utilizes collective operations, and efficiently handles data distribution and communication among threads on the distributed shared-memory architecture. This study utilises the *2-D block-cyclic* data distribution, which has been shown to be key to achieving scalable performance. In the *2-D block-cyclic* data distribution, processors operate primarily in parallel on local data. Remote data is retrieved in large blocks by each processor to optimize memory bandwidth and minimize the impact of memory latency.

A traditional UPC implementation of the matrix-matrix multiplication with a block distribution is shown in code listing 5.1 below. The implementation closely resembles traditional sequential code, making UPC straightforward to program and enabling incremental parallelization of existing sequential codes. Shared (global) arrays are declared using the keyword *shared [block-size]*, which distributes the arrays in blocks per thread in a round-robin manner.

```
shared[N] double A[N][N], B[N][N], C[N][N];
void matrixmultiply_upc()
{
    int i, j, k;
    double mysum;
    upc_forall(i = 0; i < N; i++; &A[i][0])
     for (j = 0; j < N; j++)
     {
           mysum = 0;
           for (k = 0; k < N; k++)
              sum += A[i][k] * B[k][j];
           C[i][j] = mysum;
     }
}
```

LISTING 5.1: UPC Matrix-matrix multiplication with Block distribution

5.5.2 PGAS SMBS

Considering the matrices

$$A_{M,K} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,K} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,K} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,K} \\ \vdots & \vdots & \ddots & \vdots & \\ a_{M,1} & a_{M,2} & a_{M,3} & \cdots & a_{M,K} \end{pmatrix} \text{ and } B_{K,N} = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,N} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,N} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,N} \\ \vdots & \vdots & \ddots & \vdots & \\ b_{K,1} & b_{K,2} & b_{K,3} & \cdots & b_{K,N} \end{pmatrix}$$

the matrix-matrix product given by:

$$C \leftarrow A \times B, \text{where } c_{i,j} = \sum_{k=1}^{K} a_{i,k} b_{k,j}$$

where A, B and C are dense matrices. The algorithm proceeds as follows:

1. Partition matrices **A** and **B** into $m \times n$ blocks and distribute **A** column-wise and **B** row-wise such that $A^{+}_{0,0}$ and $B^{+}_{0,0}$ are located on processor thread $P_{0,0}$, $A^{+}_{0,1}$ and $B^{+}_{1,0}$ on processor $P_{0,1}$, etc. The blocks are distributed such that, the number of columns of **A** equals the number of rows of **B** as illustrated in figure 5.3. Each processor has data affinity to local blocks of data. Particularly, for blocks of matrix **A** which are not shifted, the distribution is done such that each thread has affinity to local block data. The specific declarations used to block and distribute the **A** and **B** matrices are illustrated in Listing 5.2. The striped 2D array distribution, with a blocking factor, THREADS, is used for the **B** matrix, ensuring that the matrix (array) is allocated in contiguous blocks on the appropriate threads. This tiled layout ensures $(M \times N)/b^2$ parallelism with the worst-case $\mathcal{O}(1/b^2)$ remote accesses.

```
struct col_block {
    float tile[b][block_cols];
};
shared col_block A [M/b][K/block_cols];

struct row_block {
    float tile[block_rows][b];
};
shared [THREADS] row_block B [K/block_rows][N/b];

/* C Matrix is distributed in similar manner as A */
shared col_block C [M/b][N/block_cols];
..
```

LISTING 5.2: UPC SMBS Block definition and distribution

where M, N and K are assumed to be multiples of THREADS; the resulting blocking factor from the qualifiers is generally computed as:

$$\frac{\frac{sizeof(a)}{upc_element_sizeof(a)} + THREADS - 1}{THREADS}$$

The block decomposition provides an efficient contiguous chunk distribution for both **A** and **B** matrices, and thus reducing the number of data movements by a factor of at least two (2) for matrix **B** compared to the original Cannon matrix-matrix multiplication algorithm. UPC THREADS determine the affinity of each data element(i), thus, improving the time complexity and performance, especially given optimum block-size which fits into the higher memory hierarchy such as cache, with corresponding reduction of cache misses. Generally, an element i of a blocked data array has an affinity to thread:

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \mod THREADS$$

2. After the data partitioning, each thread then performs computation on the local blocks, following a similar approach to the distributed parallel architecture (MPI) implementation.

3. During each iteration in the computational loop, call **Reduce** on the row to get the sum of all the partial sums, i.e., $\sum_{j=0}^{col_size} c_{i,j}$. This results in the final $C_{i,j}^{+}$ block of **C**. $MYTHREAD$ (UPC thread identifier) of the processor $P_{i,j}$ which will hold the $C_{i,j}^{+}$ is computed in a manner similar to the MPI implementation as follows:

$$(i + row) \mod col_size.$$

where i is the ith iteration,
row is the row-number, and col_size is the total number of columns per row.

4. The blocks of **B** matrix are then shifted one step up with wrap-around;

5. Compute the next block multiplication, and **Reduce** the partial results per row, and repeat $m - 1$ *times*. The formal presentation of the algorithm is also illustrated in Algorithm 5.4.

Algorithm 5.4: PGAS SMBS

Data: shared $A[M][K]$, *shared* $B[K][N]$, *gridsize* $m \times n$

Result: $C[M][N]$

begin

Distribute A column-wise and B row-wise across the $m \times n$ grid in shared memory.

for $r \leftarrow 0$ **to** $m - 1$ **do**

UPCMatrixMultiply$(A_{i,k}, B_{k,j}, C^{+}_{i,j})$

ReduceMatMultPerRow$(C^{+}_{i,j}, C_{i,j})$

$j \longleftarrow (row + r) \mod p$

$C_{r,j} \longleftarrow \sum_{j=0}^{p-1} C^{+}_{r,j}$

Up-circular-shift each column of B by 1,so that $B_{i,j}$ *is overwritten by* $B_{i,(j+1) \mod p}$

return C

The blocked algorithm divides matrices **A**, **B**, and **C** into blocks of size *BLOCK_SIZE*. The algorithm performs $\frac{N}{P}$ iterations of matrix-matrix multiplication, in each iteration it computes the dot product of a block of the matrix A and a block of the matrix B, which takes $\mathcal{O}\left(\frac{N^2}{P}\right)$ operations. In addition, it shifts blocks of matrix B, which takes $\mathcal{O}\left(\frac{N^2}{P}\right)$ operations.

Generally, in the UPC shared memory model, given the declaration of the shared array below:

$$shared[b] \ < type > \ A[d0][d1]\ldots[dn]$$

array A is organised in memory in a block-cyclic scheme with a blocking factor of b. To locate an array index $\mathbf{v} = v_0, v_1, \ldots, v_{n-1}$ for an element, $A[\mathbf{v}]$, the linearized row-major index is computed as follows:

$$L(i) = v_0 \times \prod_{j=1}^{n-1} d_j + v_1 \times \prod_{j=2}^{n-1} d_j + \ldots + v_1$$

Algorithm 5.5: UPCMatrixMultiply

```
for s ← 0 to P − 1 do
    shift ← (my_col − s + P) mod P
    upc_memget(A[my_row ∗ N/P][shift ∗ N/P], A[my_row ∗ N/P][shift ∗
      N/P], N/P ∗ BLOCK_SIZE ∗ sizeof(float))
    upc_barrier()
    for i ← my_row ∗ N/P to (my_row + 1) ∗ N/P step BLOCK_SIZE do
        for j ← my_col ∗ N/P to (my_col + 1) ∗ N/P step BLOCK_SIZE do
            for k ← s ∗ N/P to (s + 1) ∗ N/P step BLOCK_SIZE do
                for ii ← i to i + BLOCK_SIZE do
                    for jj ← j to j + BLOCK_SIZE do
                        for kk ← k to k + BLOCK_SIZE do
                            C[ii][jj] ← C[ii][jj] + A[ii][kk] ∗ B[kk][jj]
```

This linearized index underpins the block-cyclic arrangement, and the UPC thread on which the array element $A[\mathbf{v}]$ resides can be computed. Within a thread's local storage, the array is stored as a collection of blocks. The *course* of an array element refers to the block number where the element is located, while the *phase* specifies its position within that block [123].

$$\begin{cases} thread(A, \mathbf{v}) & ::= \left\lfloor \dfrac{L(\mathbf{v})}{b} \right\rfloor \mod \mathcal{T} \\ phase(A, \mathbf{v}) & ::= L(\mathbf{v}) \mod b \\ course(A, \mathbf{v}) & ::= \left\lfloor \dfrac{L(\mathbf{v})}{b \times T} \right\rfloor \end{cases}$$

It is worth noting that the size of each block is a trade-off between the amount of computation and the amount of data movement, the optimal size of the block will depend on the specific system. The optimal block size depends on various factors, such as the cache size and associativity of the processor, the memory access patterns, and the size of the data. Choosing blocks which reduces cache-misses and increases data-locality, ensures a higher algorithmic performance. In this implementation, there is no need for explicit block transfers as each thread has access to the shared array, though, threads have affinity to local blocks. Regardless of the affinity of the data, all processors have access to any location within the shared-space array using traditional array index operations. The underlying runtime mechanisms generate the necessary communication and remote calls.

5.6 Performance Evaluation

5.6.1 Experimental setup

The study investigated the performance of the SMBS algorithm with the Cannon and variants of the SRUMMA algorithms included in the Global Arrays Toolkit, on the Edison cluster of the National Energy Research Scientific Computing Center (NERSC). The experiments were conducted using up to 2500 cores of the cluster. Each selected cluster node has two sockets, with a 12-core 2.4 GHz Intel Xeon CPU E5-2695 v2 (“Ivy Bridge”) processor, a total of 24 cores and 64 GB DDR3 1866 MHz memory (four 8 GB DIMMs per socket) per node. The nodes are interconnected in a Dragonfly topology with the Cray Aries. All the codes were written in C and compiled using Intel MPI. Each of the experiments was conducted using random non-zero synthetic double-precision floating-point numbers. The Intel MPI 5.0 library and the Global-Array toolkit GA-5.2 were used.

Another set of experiments was conducted to assess the performance of the distributed shared-memory implementation of SMBS with a tuned state-of-the-art SRUMMA matrix-matrix multiplication from the Global Array toolkit 5.2. These performance evaluations were conducted on the Centre for High Performance Computing (CHPC), South Africa, Lengau Cluster. This petascale system consists of Dell servers, powered by 5th generation Intel processors with 56 Gbps FDR Infiniband interconnect. Each compute node

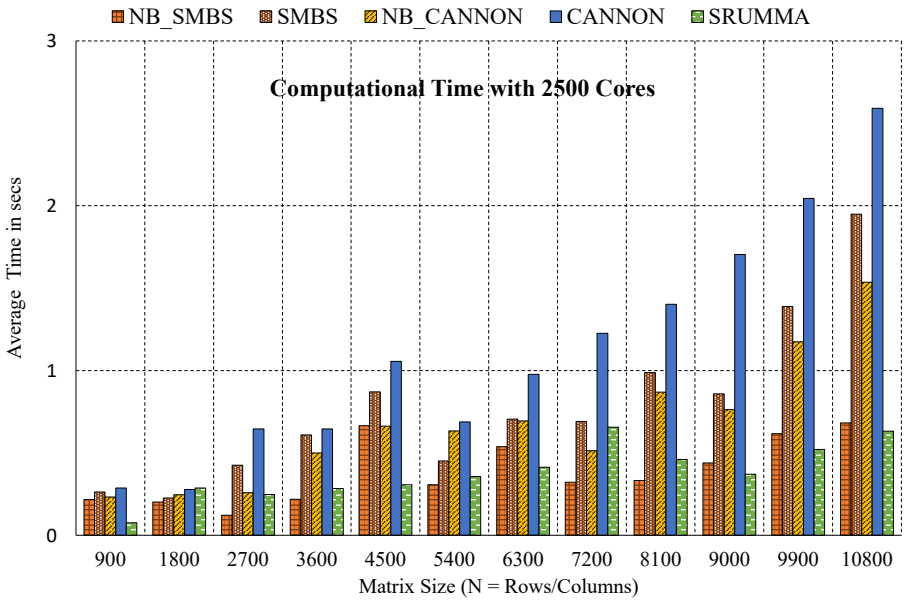
comprises of 24 2.6 GHZ Intel Xeon cores, with 64GB of memory. The study evaluated the algorithms using up to 256 CPU cores, and use dense matrices generated from random non-zero synthetic double-precision floating-point numbers as well as the real-world image dataset, CIFAR-100 (Canadian Institute for Advanced Research, 100 classes) [124]. CIFAR-100 is a benchmark dataset created by the Canadian Institute for Advanced Research, and widely used in machine learning and computer vision research. They are crucial for developing and testing various image recognition and classification algorithms. It consists of a labelled subset of 80 million tiny images' dataset, including, 60000 32×32 colour images. The CIFAR-100 dataset consists of 100 classes grouped into 20 superclasses, with each class containing 600 images. Each image is assigned a "fine" label (indicating its specific class) and a "coarse" label (indicating its superclass). 500 of the images are marked as training datasets, and 100 as testing dataset per class. The CIFAR-100 dataset contains a diverse range of objects and scenes, reflecting various real-world scenarios that algorithms might encounter. This study extracted the images from the CIFAR-100 dataset and converted them to their matrix representations to provide a real-world application dataset.

For each test, the algorithm was run three (3) times and the average time (given as the difference between the start and end times) reported. The time was reported using the *MPI_Wtime()* (for MPI implementations), as well as the *upc_ticks_now()* and *UPC_TICKS_PER_SEC* (for UPC implementations). These provide wall-clock time in seconds with high precision to capture the computation time for each run.

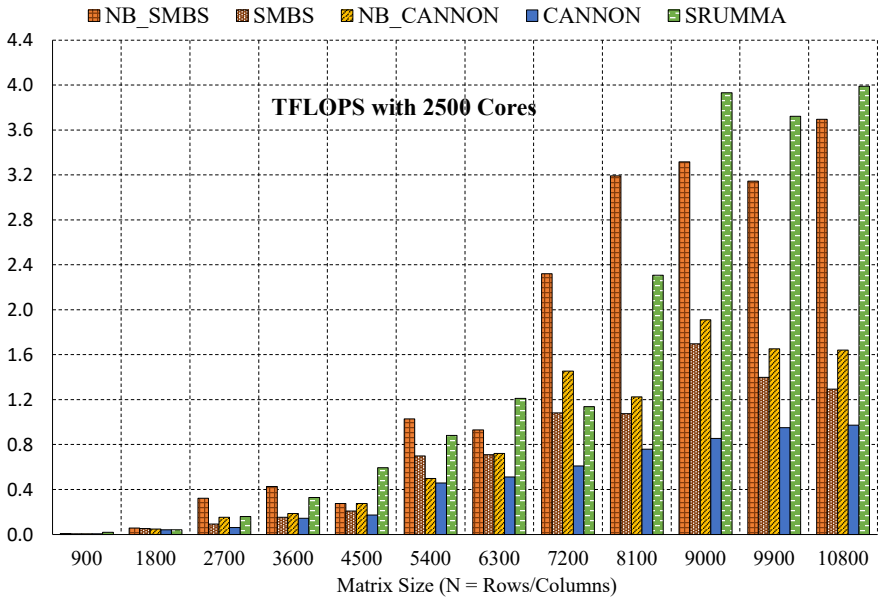
5.6.2 Result And Discussion

The study conducted several experiments to evaluate the efficiency of SMBS in comparison to the Cannon and SRUMMA algorithms, assessing both blocking and non-blocking implementations. The first test involved increasing the matrix sizes $(N \times M)$ from 900×900 to $10,800 \times 10,800$ while the number of nodes and processing cores remained the same. For simplicity in the evaluations, N was set to $N = M$ in all experiments, making both matrices **A** and **B** of sizes $N \times N$. The time to compute the solution $\mathbf{C} \longleftarrow \mathbf{A} \times \mathbf{B}$ for each of the Cannon and variants of the SRUMMA algorithms was

compared to SMBS. The primary goal in these tests was to evaluate the effect of distributing across several nodes, in which case inter-node latencies will affect the overall computational throughput. Additional sets of experiments were conducted to characterize the scalability of the algorithms given a different number of processing cores and workload sizes.



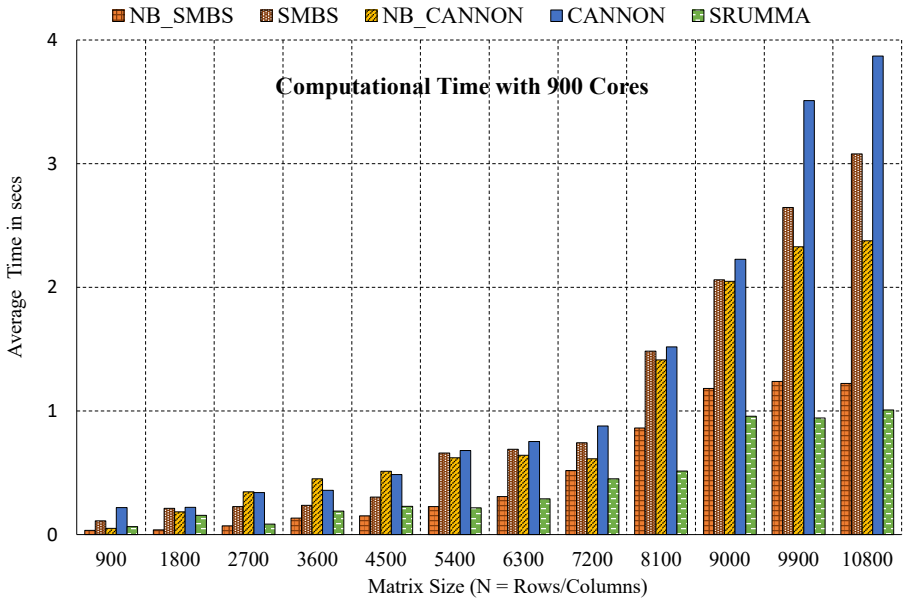
(A)



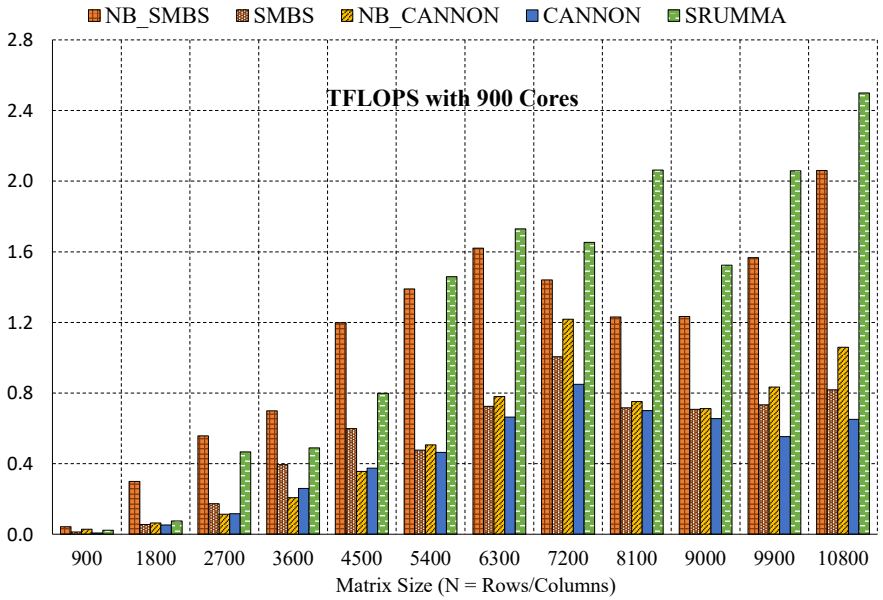
(B)

FIGURE 5.5: Computational time and Operational throughput for various Matrix Sizes using 2500 cores

Figures 5.5a, 5.6a and 5.7a present the average time to compute the solution, while



(A)

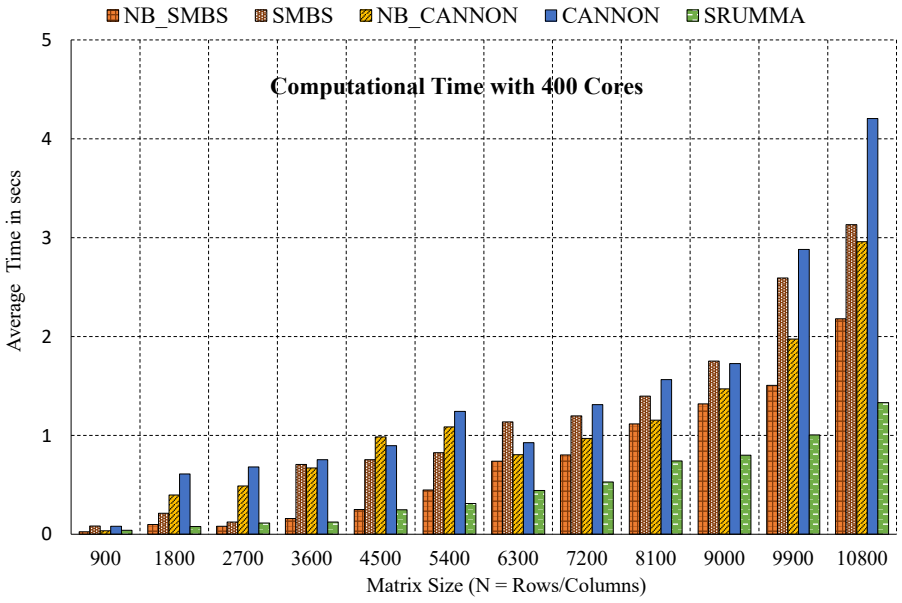


(B)

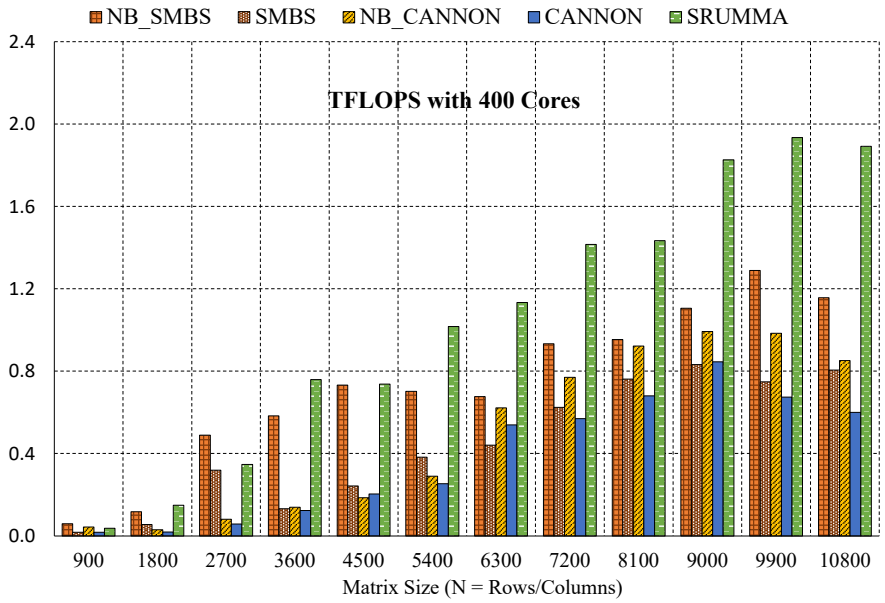
FIGURE 5.6: Computational time and Operational throughput for various Matrix Sizes using 900 cores

Figures 5.5b, 5.6b and 5.7b present the corresponding operational throughput in *teraflops (TFLOPS)*. Each test was repeated 3 times and the average time, and throughput recorded. The matrix sizes used are the squares of those shown in the figures (*i.e., 900 corresponds to a* 900×900 *matrix, 1800: refers to a* 1800×1800 *matrix*, etc.).

Figure 5.5a highlights a general increase in computation time as the size of the matrices increases for all the algorithms. As the matrix sizes increase, more resources



(A)



(B)

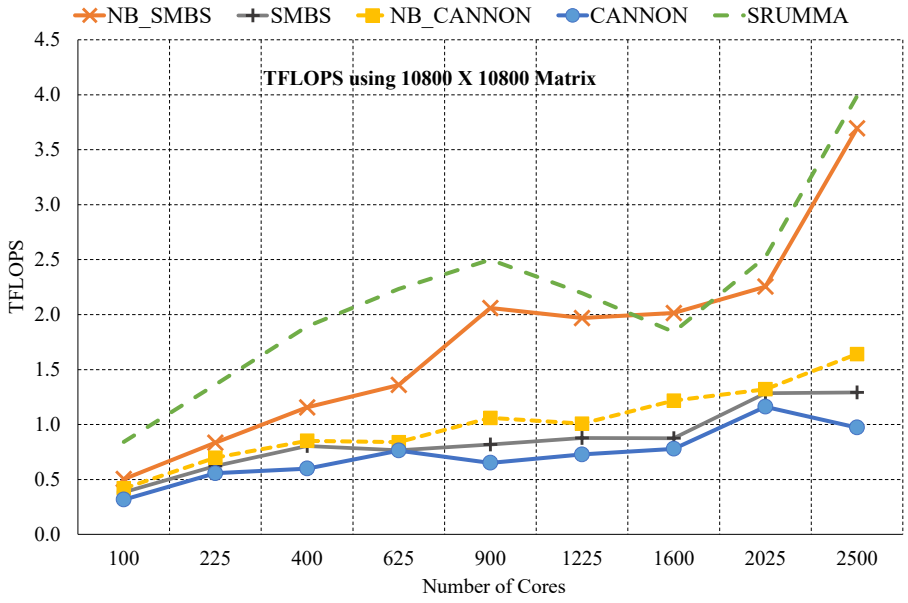
FIGURE 5.7: Computational time and Operational throughput for various Matrix Sizes using 400 cores

are required by each processor, since the corresponding block sizes per core also increase. This results in more communication and computation between nodes, and subsequently resulting in increased latencies. SMBS performs better than Cannon since it does fewer data movement and hence less inter-node communication. The non-blocking SMBS (referred to as NB_SMBS) shows a much better performance improvement compared to the non-blocking implementation of the Cannon algorithm (referred to as NB_CANNON). As the sizes of the matrices increase, the rate of increase in the

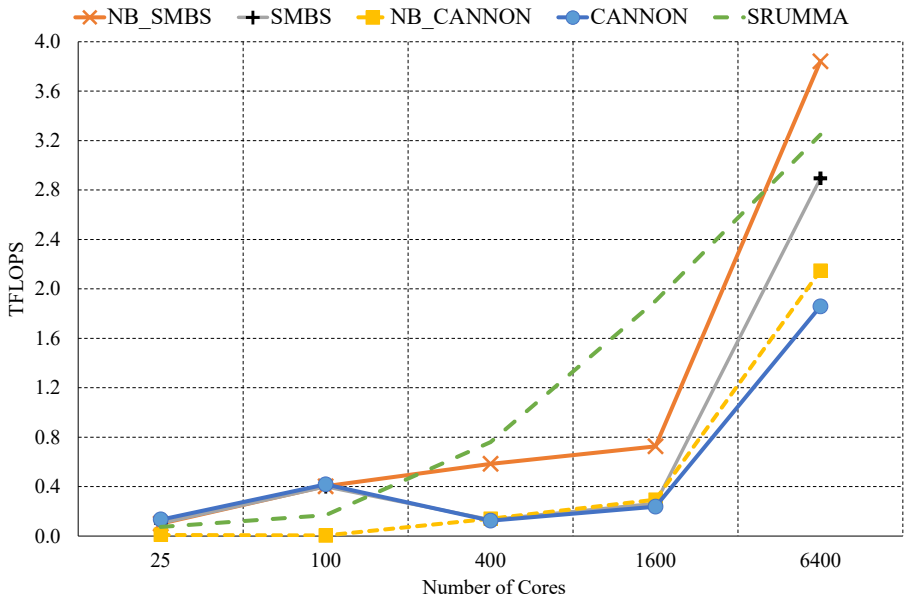
latency is much lower in the SMBS than in the Cannon's algorithm. Both non-blocking algorithms have significantly lower latencies (close to 50% for NB_SMBS and 60% for NB_CANNON) as expected, with NB_CANNON growing much faster in terms of the worst latency. This is primarily because as the matrix sizes increase, so does the block sizes and the subsequent increase in the time to perform block shifts. Although the non-blocking implementations overlap computation with data movement, the overhead cost incurred is obvious from the experimental results, particularly for NB_CANNON. NB_SMBS does not incur much latency in data movement since data movement is limited to only a single matrix, whiles employing latency hiding by overlapping the data communication with the computation and hence the better performance recorded. Further, unlike the NB_CANNON, NB_SMBS takes advantage of data locality since the A matrix is not shifted during the data shifting steps. This naturally translates into the higher FLOP values as shown in Figures 5.5b, 5.6b and 5.7b. Furthermore, it was noted that NB_SMBS performed comparably to the highly tuned SRUMMA, outperforming SRUMMA for some workloads. NB_SMBS outperformed the NB_CANNON in all workloads as anticipated. Generally, more cores will result in fewer blocks per core, as well as faster data transfers and computations and effectively lower latencies.

The next set of experiments were conducted to determine the scaling efficiency (scalability) of SMBS, especially the non-blocking SMBS, compared to Cannon and SRUMMA. In the strong scaling test, the experiment evaluates how the computational time of a fixed-size matrix changes when the processor resources increase. The goal of the strong scaling test is to assess the efficiency of the algorithm when additional computational resources are allocated. The weak scaling test, on the other hand, measures how the time to compute the product varies with processor count with a fixed system size per processor.

In the strong scaling test, the workload was fixed at 10800×10800 while increasing the processing cores. The average FLOPS for three (3) runs was computed after increasing the processor cores from 100 to 2500 cores. Figure 5.8a shows the graph for the strong scaling efficiency test. As the number of processor cores increases, less work is required by each core/worker, and therefore the computational time reduces, and the operational throughput for that matter increases. Operational throughput generally



(A) Strong scale



(B) Weak scale

FIGURE 5.8: Evaluation of algorithms' scaling efficiency. This experiment kept a fixed problem size of 10800 × 10800 and increased the number of processors from 100 to 2500 cores in the strong scale test. In the weak scale test, the workload was increased from 60 × 60 to 14400 × 14400 matrices, while the processors increased from 25 to 6400 cores

increases as the core count is increased. However, a slow increase is observed in the blocking algorithms. This could be attributed to an increase in inter-process communication latencies which is, however, overlapped with computation resulting in good scaling efficiency for the non-blocking algorithms.

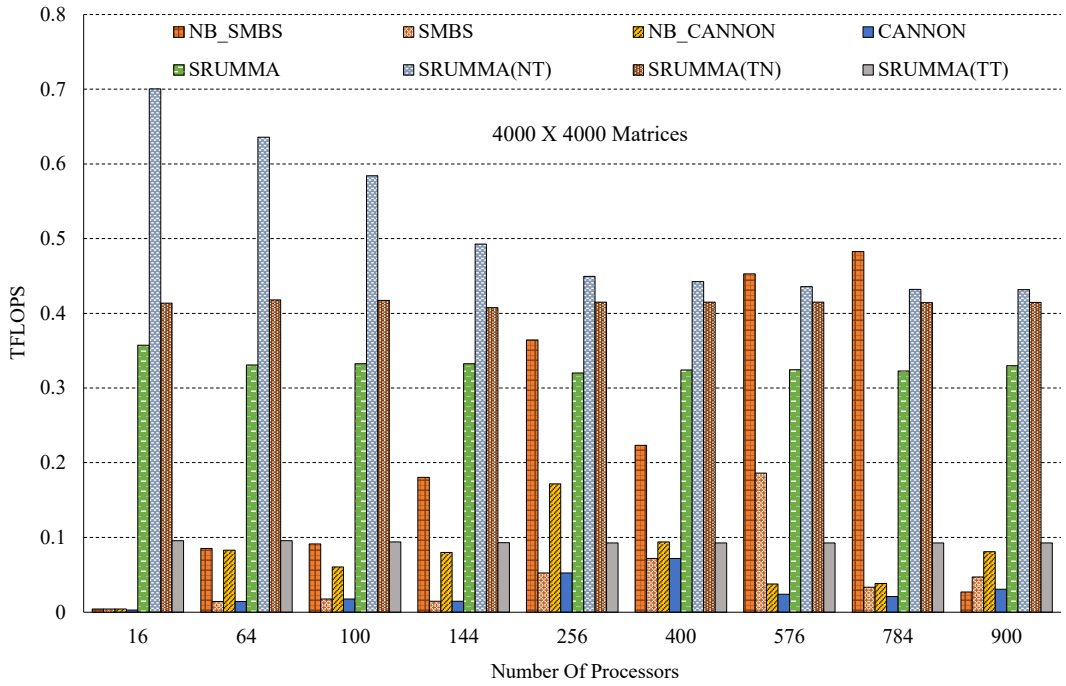


FIGURE 5.9: Scaling efficiency using 4000 × 4000 Matrix with Variants of SRUMMA

Additionally, Figure 5.9 further shows the results of evaluating the scaling efficiency of SMBS compared to the Cannon algorithm and SRUMMA variants. Variants of the SRUMMA used here refer to various matrix transposition configuration options for the Global Array (GA) PDGEMM algorithm. In Figure 5.9, this section refers to SRUMMA as the configuration with no matrix transpositions, SRUMMA(NT): when the **B** matrix is transposed, SRUMMA(TN): when the **A** matrix is transposed and SRUMMA(TT): when both **A** and **B** matrices are transposed. In this strong scaling test, the number of cores was increased from 16 to 900 cores with a workload of 4000 × 4000 matrix. The figure highlights similar results observed in Figure 5.8a with NB_SMBS demonstrating comparable performance as the number of processor cores increased, and outperforming SRUMMA (NT, TN, and TT) in some instances. It was also noted that, the performance gap between NB_SMBS and NB_CANNON increases significantly as the number of cores increase. This is because more cores require more inter-node communication. NB_SMBS minimises the amount of such communications and consequently translates to a relatively better performance than the Cannon algorithm.

Figure 5.8b shows the results of the operational throughput in TFLOPS for the weak

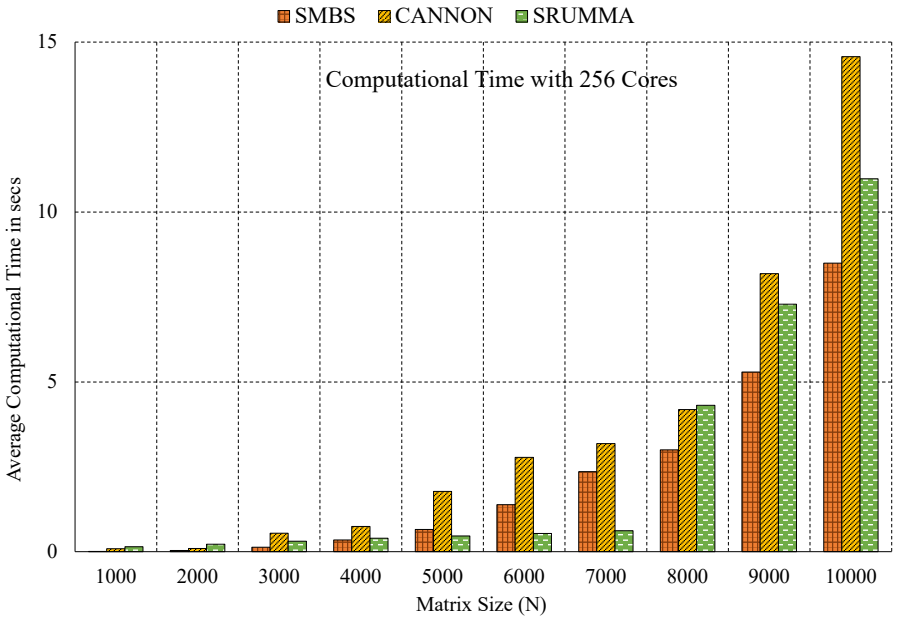
scaling test in which the workload (matrix size) is increased from 60×60 to 14400×14400 while increasing the processor cores from 25 to 6400. The test shows a general increase in performance as both cores and problem sizes increase. Performance gains increased at a much higher rate when $P = 1600$ (corresponding to matrix size of 3600×3600) in most cases, though there was a gradual increase from the initial cases. SRUMMA did perform best of all, however, NB_SMBS demonstrated a comparable performance as the number of processor cores increased with load. The NB_SMBS algorithm performed better than Cannon's algorithm as the processor/load increased.

5.6.3 EPGASS SMBS

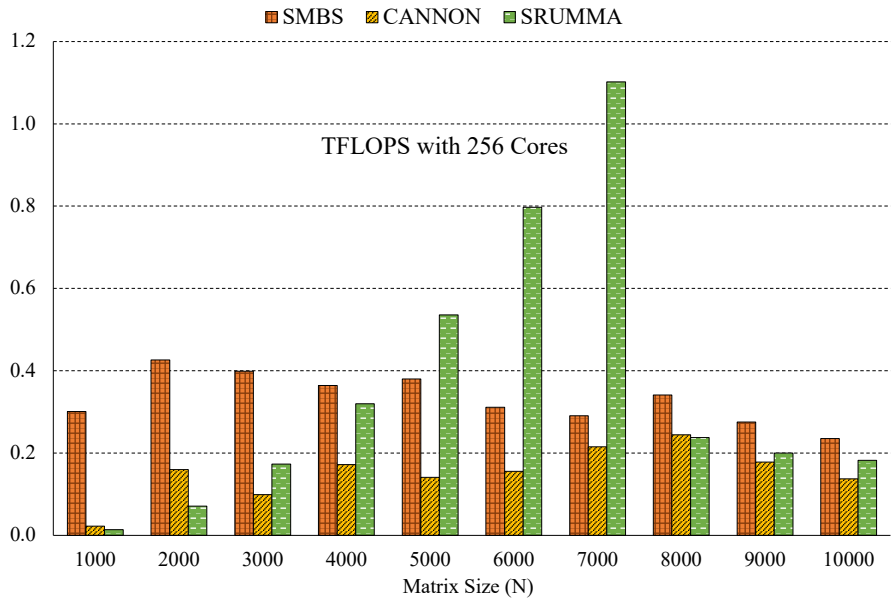
This study evaluated the distributed shared-memory implementation of the algorithm using UPC. Experiments were conducted to test the algorithm with both synthetic and real-world matrix datasets compared to that of the well-known SRUMMA from the global array toolkit and UPC versions of the Cannon algorithm. The study subsequently investigated how the algorithms scaled with different cores and workloads, including the assessment of inter-node latencies, which affect computational times.

The initial set of experiments in this group varied the matrix size $(N \times M)$ from 900×900 through to $10,800 \times 10,800$ in a manner similar to previous experiments, while keeping the number of nodes and processor cores constant. The average times taken to compute the solution $\mathbf{C} \longleftarrow \mathbf{A} \times \mathbf{B}$ for the algorithms were recorded and compared.

Figures 5.10a and 5.11a show the average times to compute the solution, and Figures 5.10b and 5.11b show their corresponding throughputs given varying matrix sizes. Computational time was found to increase with matrix size for all the algorithms considered. As the sizes of the matrices increase, there is more workload for the processors. The block sizes per core increase, resulting in more inter-node communication, as well as increased latencies. SMBS performs better than the Cannon since it does relatively fewer remote accesses, and effectively less inter-node communication. As the sizes of the matrices increase, the rate of increase in the latency is relatively lower in the SMBS compared to Cannons' algorithm. Both algorithms show significantly lower latencies, as expected, compared to the earlier message passing implementations. The distributed shared-memory implementation does not involve explicit data transfers,



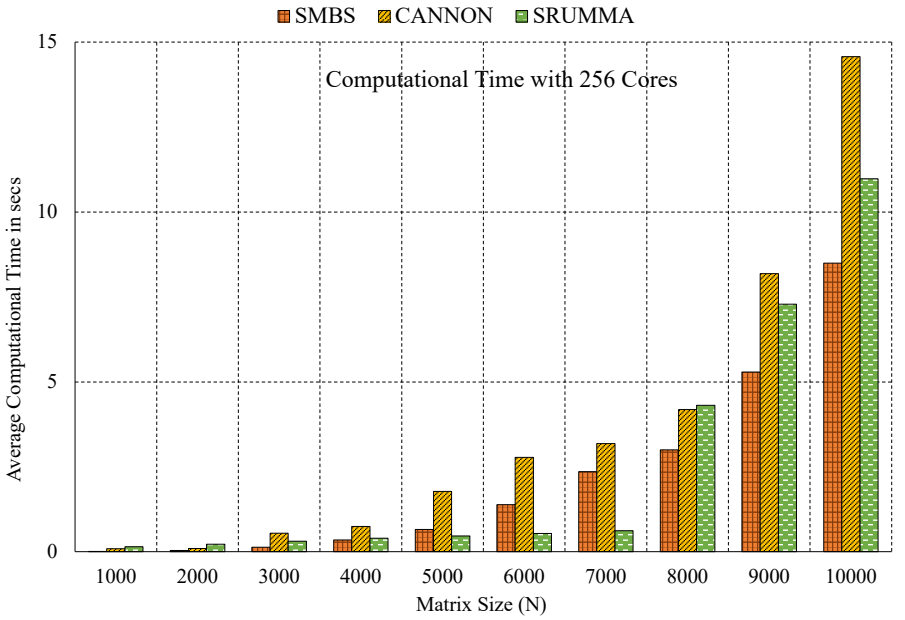
(A)



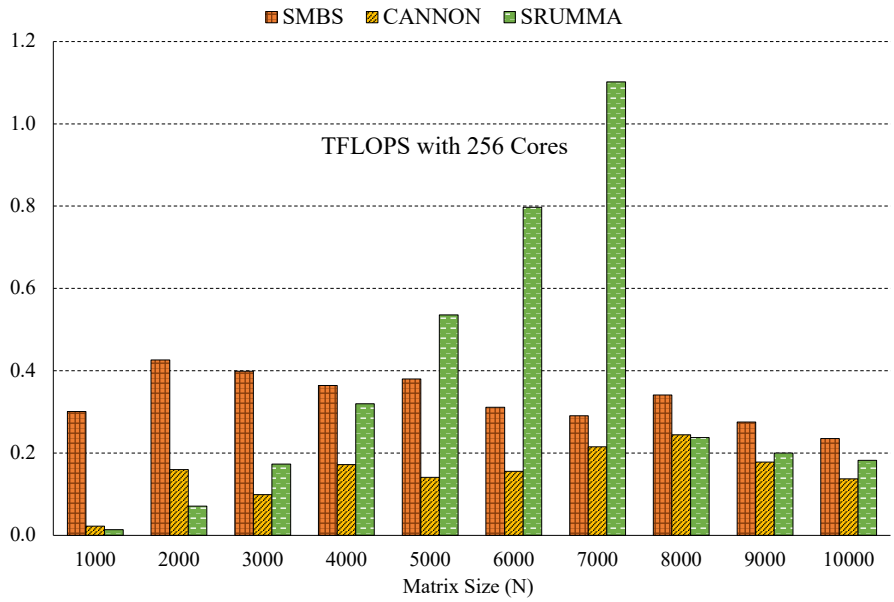
(B)

FIGURE 5.10: Average time and throughput for various Matrix workloads for 256 cores

but leverages the underling RMA capabilities, including data locality, which further enhances algorithmic performance. This translates into the relatively higher FLOP values shown in Figures 5.10b and 5.11b. SMBS demonstrates a comparable performance to the SRUMMA. The SRUMMA algorithm from the Global array library (*ga_dgemm*) adopts the distributed shared memory, including a tuned blocked allocation technique



(A)

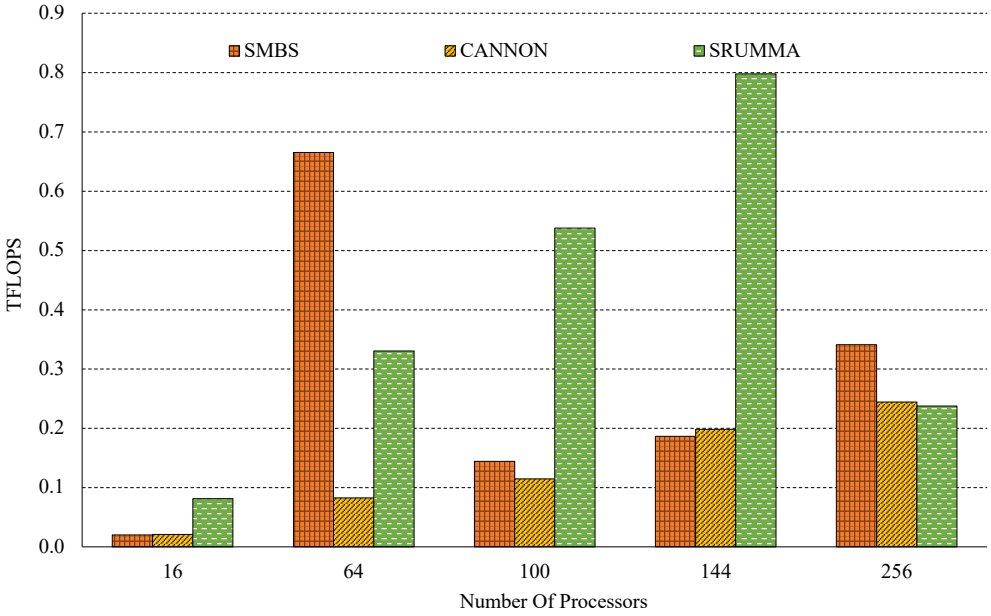


(B)

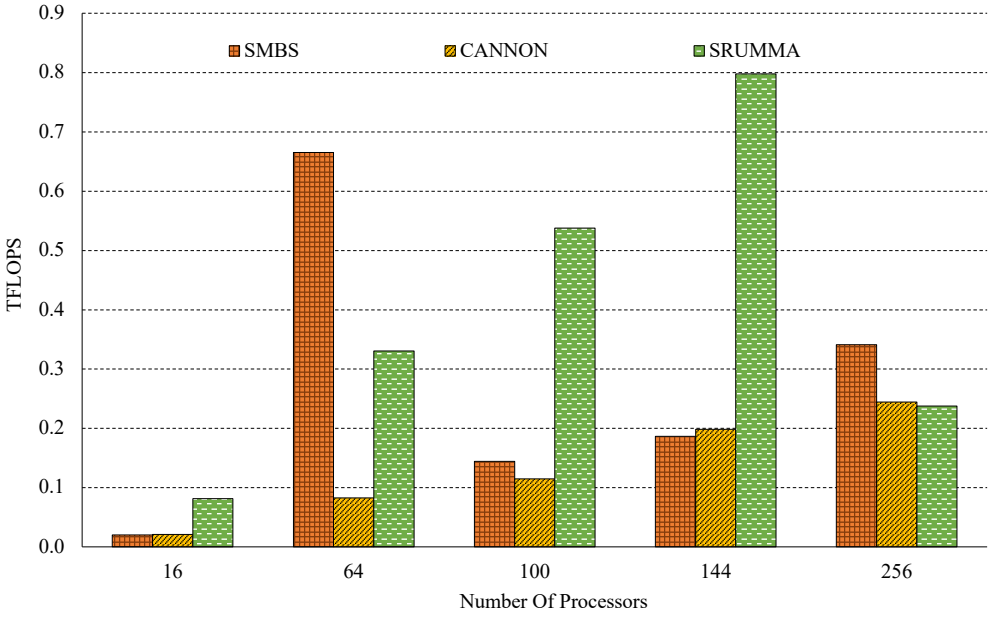
FIGURE 5.11: Average time and throughput for various Matrix workloads for 196 cores

to optimise performance, though it requires some additional memory. SMBS was observed to outperform SRUMMA for certain workloads and processor core counts.

The figures in 5.12a and 5.12b illustrate the comparative scaling efficiency of the algorithms discussed. In the strong scaling tests, the experiments maintained a fixed workload of 10800×10800, while the number of processing cores was increased from 16 to 256, with average throughput subsequently computed. As the number of cores



(A) Strong scale



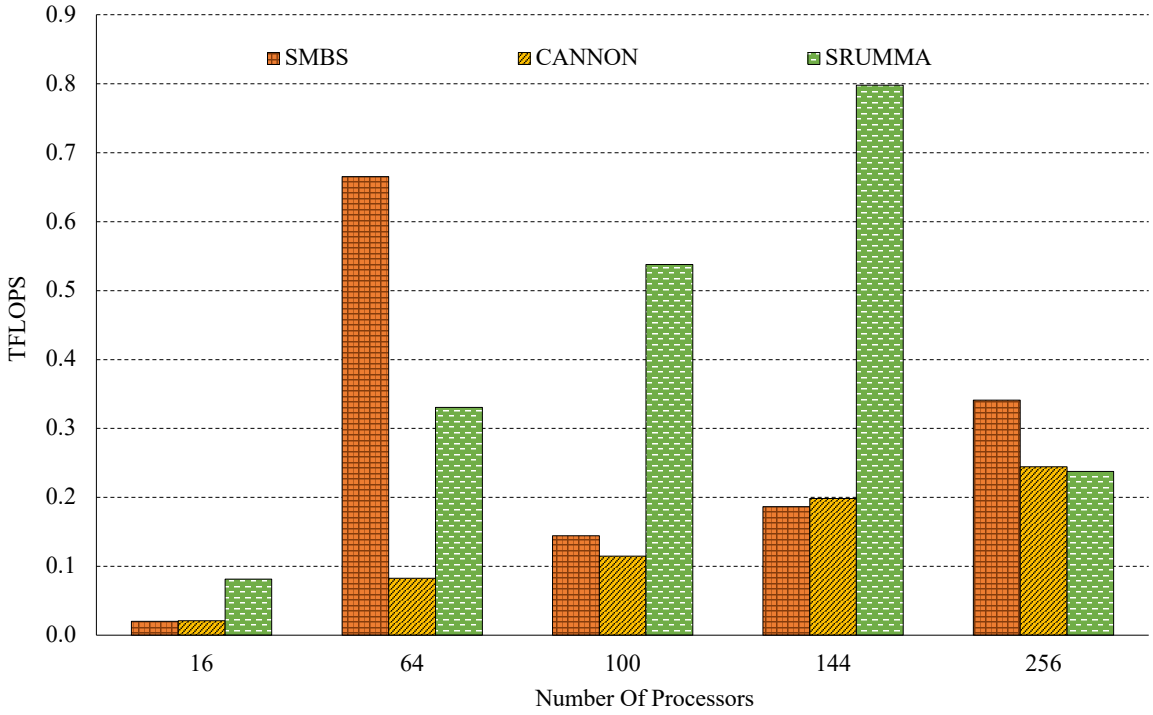
(B) Weak scale

FIGURE 5.12: Evaluation of algorithms' scaling efficiency. The strong scale test maintains a fixed problem size of 10800×10800, while the number of processors is increased to 256 cores. For the weak scale test, however, the workload is increased from 60×60 to 14400×14400 matrices, while the number of processors are doubled from 16 cores up to 256 cores

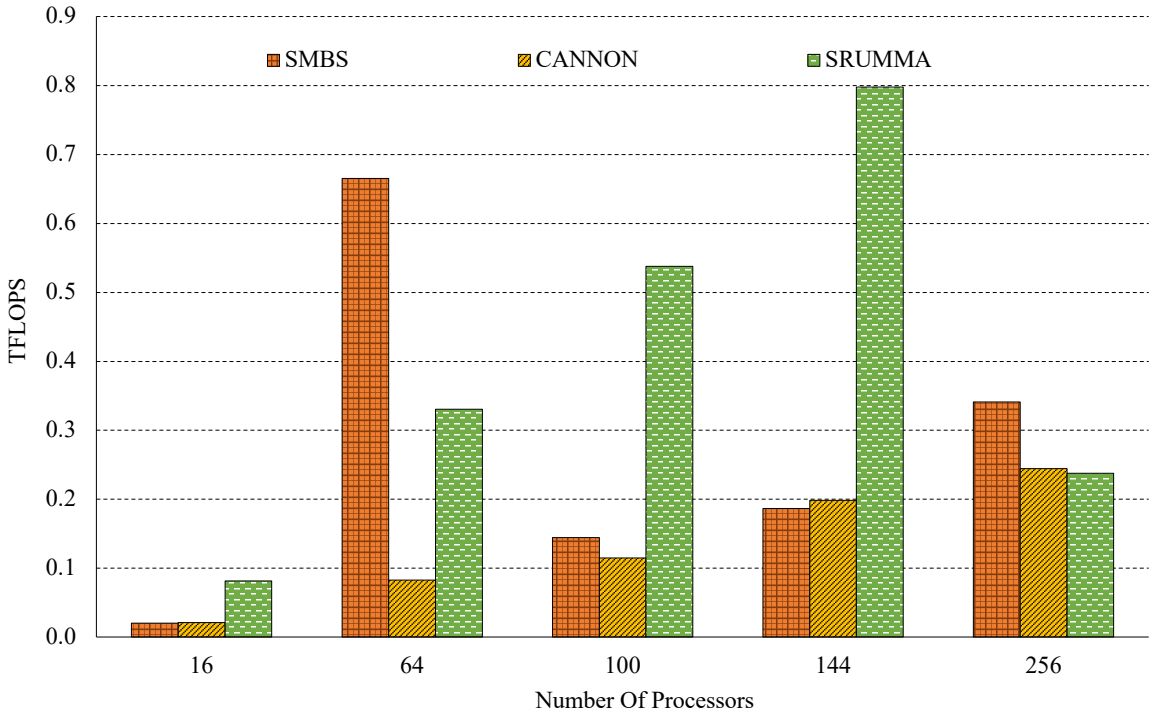
increase, less work is required by each core and therefore the computational time reduces and operational throughput increases. The operational throughput generally increases as the number of cores is increased, though, the inter-node communication and

remote data accesses increase as well. However, overlapping the computation with the communication in the implementations results in good scaling efficiency for the algorithms. Figure 5.8b shows the results of the throughput for the weak scaling scenario in which the matrix sizes are increased from 60×60 to 14400×14400 while increasing the processor cores from 16 to 256. The test shows a general increase in performance as both cores and problem sizes increase. SRUMMA did perform best of all, however, SMBS had a comparable performance as the number of processor cores increased with load. SMBS demonstrated throughput performance improvements of about $1.15\times$ over SRUMMA at 6400 cores. As the processor/load increased, SMBS outperformed Cannon's algorithm as expected and demonstrates a performance comparable to that of the SRUMMA algorithm.

Finally, figures 5.13b, and 5.13a highlight the scaling efficiency and throughput of the algorithm using a real-world image processing dataset, showing performance gains with increasing processing resources. The extracted image datasets were converted to their representative matrices $\mathbf{M}$, and the solution $\mathbf{C}$ computed such that: $\mathbf{C} = \mathbf{M} \times \mathbf{M}$. The images from the CIFAR-100 datasets are effectively represented as 3D tensors $(height \times width \times colour\ channels)$. For simplicity, this study uses a single colour (grayscale) channel represented as a 2D matrix. The elements of the resulting matrix, $\mathbf{C}$, represent the interaction between pixel values across the image. Each element encapsulates a blend of intensities from different parts of the image. This is used in some advanced image processing techniques, like convolution operations, where filters or kernels are applied to extract features. These concepts are often used in machine learning and computer vision problems, where filters consisting of small matrices are multiplied with the image patches to uncover relevant features [125–127]. Similarly, multiplying these matrices by themselves (or with their transposes) could also be used in Principal Component Analysis (PCA) and other techniques to analyse image properties, such as computing the covariance matrix to understand variance and correlation in pixel intensities [128, 129]. Appendix Figure A.1 illustrates the application of a 3×3 matrix filter/kernel to a grayscale input image in a layer of a Convolutional Neural Network (CNN) to obtain the convolved feature, which is passed on to the next layer in the network. CNN is a specialized type of deep learning model designed to process data with



(A) Strong scaling test



(B) Weak scaling test

FIGURE 5.13: Evaluation of algorithms' scaling efficiency with CFAIR-100 image processing dataset

grid-like topology, such as images [130]. They are particularly effective in image and video recognition, recommender systems, as well as natural language processing, using convolutional layers to adaptively learn spatial hierarchies of features from input data.

5.7 Chapter Summary

Many algorithms for scientific and numerical computations employ matrix-matrix multiplication as a fundamental operation, serving as the foundation for numerous operations used in numerical solvers and graph theory problems. Several algorithms (such as Cannon, PUMMA, SUMMA, etc.) have been proposed for the implementation of matrix-matrix multiplication on distributed memory architectures, and these have been extensively studied. One objective of distributed in-memory processing is to leverage distributed computing and memory resources to enhance algorithmic performance, particularly for large datasets.

This study has presented a Single-Matrix-Block Shift distributed matrix-matrix multiplication algorithm which scales well with increasing number of cores and provides a better performance than the existing Cannon's algorithm by up to $\approx 2.3\times$ and performs comparatively to a well-tuned SRUMMA algorithm. SMBS achieves higher performance by limiting the number of block shifts and essentially reducing the amount of data movement (Shifts) in the traditional Cannon algorithm. SMBS presents an elegant approach to obtain final block results per computational loop unlike Cannon's and other well-known algorithms like SRUMMA, LaPACK, etc., which present partial results until the final reduction step is complete. SMBS eliminates the final reduction step as the final results become available per iteration. This makes SMBS very applicable to image processing applications, for instance, which require several portions of the final product while processing continues.

Communication has been identified to contribute more than half of the total latency in large distributed algorithms. Thus, a technique to minimize data movement and inherently exploit data locality is much desirable. SMBS consequently reduces the amount of communication and data movement during computation. Matrix-matrix multiplication, which is a foundational routine in many scientific applications, can leverage the

enhancement shown by SMBS for further performance gains and processing throughput. The study further introduces a non-blocking SMBS implementation that employs non-blocking point-to-point messaging to overlap communication with computation, yielding performance improvements exceeding $5\times$ relative to the blocking SMBS in dense matrix–matrix multiplication.

Furthermore, this study presented the implementation of SMBS within the PGAS paradigm and evaluates its performance using synthetic datasets as well as real-world datasets for image processing and classification in deep learning. The matrix-matrix multiplication of the real-world CFAIR-100 datasets presents concepts adopted in some advanced techniques, like convolution operations in convolutional neural networks, as well as dimensionality reduction and feature extraction. The study has shown from the experimental results that exploiting data locality, and cache coherency subsequently results in improved algorithmic performance. The implementation introduces an efficient technique for accessing data blocks across distributed memory, with potential for further tuning to improve performance. Providing a shared memory space that all processors can access abstracts the complexity of direct message passing, significantly reducing the cost associated with establishing and managing communication channels.

Chapter 6

k-d Trees And EPGASS

6.1 Introduction

The *k*-d tree is a data structure that is used to organize points in a *k*-dimensional space by recursively partitioning them along alternating axes, and thus extends the concept of a binary search tree to accommodate multiple dimensions. Each leaf node represents a point in the *k*-dimensional space. A non-leaf node in a *k*-d tree partitions the space into two regions, known as *half-spaces*, with the left subtree representing points in the left subtree and the right subtree representing points in the right subtree. By separating space through split regions, nearest-neighbour searches, for example, can be made much faster when using an algorithm such as the Euclidean clustering. Generally, for the planes $0, 1, 2, \ldots (k-1)$, a data point at a depth $\mathbf{d}$, will have a plane $\boldsymbol{P}$, which is calculated as: $\boldsymbol{P} = \mathrm{d} \mod \mathbf{k}$. If the root node is aligned with plane $\boldsymbol{P}$, the left subtree will consist of all points with coordinates in that plane smaller than the root node's coordinate. Conversely, the right subtree will include all points with coordinates in that plane greater than or equal to the root node's coordinate.

Given a set of spatial search queries in Euclidean space, illustrated in Table 6.1, a brute-force search would perform them by inspecting all the data points in the set. When performing m queries on a set of n data points, this could result in $\mathcal{O}(mn)$ operating time, which would be inefficient for many applications. Hyperplanes that are orthogonal to one of the coordinate axes are used to define a recursive partitioning of the space.

The spatial queries presented in Table 6.1 can be accelerated and executed more efficiently by focusing solely on the points within the relevant partitions when the search space is structured into geometric regions. The k-d tree has been widely researched, leading to the development of many variants, primarily distinguished by the methods used to determine the splitting planes [131–133]. For example, the spatial median heuristic divides the space at the midpoint of the widest dimension of the current node. More sophisticated heuristics, including the Surface Area Heuristics (SAH), choose to use a splitting plane by characterizing the spatial distribution of a node's data points. Unlike the spatial median, the SAH k-d tree [134, 135] attempts to reduce the running time for the nearest neighbour and related traversals. SAH chooses splitting planes by minimizing a defined cost function $C(S)$.

6.2 Background

Over the past years, k-d trees have received considerable attention, especially in building the tree index for efficient traversals with optimized low-level implementation and memory layout. The construction of an efficient k-d tree (with n elements) can be time-consuming, with a time complexity that could typically range from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n^2)$.

TABLE 6.1: Sample queries in space: where P is a set of data points in $\mathbb{R}^k$ space, and $\mathbf{q}$ a search/query point in $\mathbb{R}^k$.

		Definition
1	Nearest neighbour	$\text{argmin}_{p \in P} \|p - q\|$
2	k-NN	$X \subseteq \mathcal{P}$ s.t. $\|X\| = K$ and $\|\mathbf{x} - \mathbf{q}\| \leq \|\mathbf{y} - \mathbf{q}\|$ for any $\mathbf{x} \in X$ and $\mathbf{y} \in \mathcal{P} \setminus X$
3	Range query	$\{p \in \mathcal{P} \mid \|\mathbf{p} - \mathbf{q}\| < r\}$

While the k-d tree can significantly accelerate searches compared to brute-force methods, the construction time often remains a drawback in many applications. Parallelism in modern processors and high-performance infrastructure offers opportunities for further speed-ups and optimizations. Though traversals are relatively easy to parallelize over a large number of queries, the tree construction and balancing is not as easily parallelized, and presents a serial constraint to applications if not parallelized. Unlike tree

indexes such as the Red–Black trees, AVL trees, and B-trees, the *k*-d tree construction, particularly keeping the tree balanced, is a very challenging task since the structure cycles through different dimensions at each level of the tree. There have been several attempts to parallelise and scale the construction of the *k*-d tree to optimise performance [132, 133, 136, 137].

The primary objective of many previous works was to be able to process large-scale, multidimensional datasets more effectively, particularly for *k*-nearest neighbour (*k*-NN) searches. Patwary et al. [136] adopted a distributed tree construction approach using message passing that involved global and local tree construction mechanisms where the underlying data is split and distributed among the available nodes, which construct local *k*-d trees independently. A *k*-NN search with their implementation results in the traversal of the global tree to identify the node that owns the domain containing the query, which then receives the query for further processing [136]. Their approach ensures efficient data partitioning and load balance to enhance query results.

The Fast Library for Approximate Nearest Neighbours (FLANN) [138, 139] is one of the state-of-the-art libraries which contains a collection of optimised algorithms for performing fast approximate nearest neighbour searches in high dimensional spaces. The library includes several *k*-d tree implementations which have been incorporated into various machine learning and computer vision libraries such as the Point Cloud Library (PCL) [140, 141], and the Vision Lab Features (VLFeat) [142] library for efficient spatial searches. The FLANN library includes the Randomized *k*-d tree implementation which constructs multiple *k*-d trees with random variations, which increases the algorithm's probability of finding the nearest neighbours quickly. This approach, however, uses the approximate nearest neighbour searches, which strikes a balance between accuracy and computational speed.

In this chapter, the thesis explores and presents an alternate parallel *k*-d tree construction algorithm, leveraging the EPGASS in-memory data storage technique. The study uses a distributed approach similar to that of Patwary et al. [136] to optimise the tree

construction. However, a distributed-memory approach with no explicit message passing is adopted, including a novel data partitioning approach which enhances data locality. Additionally, the chapter presents the experimental comparative evaluations of the tree construction as well as nearest neighbour searches with the Fast Library for Approximate Nearest Neighbours (FLANN).

6.3 Application of *k*-d trees

k-d trees are widely used in several applications, including graph partitioning, hierarchical applications as well as databases. All of these applications may benefit from efficient and fast balanced *k*-d tree construction. Some specific use cases include:

- Ray tracing and rendering in computer graphics to efficiently locate the nearest surfaces intersected by rays, which is critical in generating realistic images.
- 2D range search, such as querying all points within a given range or hyper-rectangle commonly used in database searches, geographic information systems (GIS), and multidimensional indexing.
- Spatial databases to efficiently index spatial data for the fast retrieval of spatial objects such as points, lines, and polygons.
- Nearest-neighbour query to locate the closest point(s) to a given point in multi-dimensional space in many areas, such as computer vision, robotics, and pattern recognition.
- Machine learning, in classification and regression problems, *k*-d trees are used for efficient k-nearest neighbour searches, which are often part of the model training and prediction processes.
- Collision detection, in robotic motion planning to find nearest neighbours for obstacle avoidance and pathfinding.
- Flight simulators for spatial indexing and efficient search of 3D space. They are particularly useful for handling large datasets and performing fast queries, which are essential for real-time rendering and interactive elements of flight simulators.

- Several vision algorithms conduct searches on an existing collection of images. For example, searching through millions of photos to identify possible replacements to fill missing image regions; labelling images based on similarity with pre-labelled images in a repository; object detection, and many more. The similarity between entire images can be evaluated using the Euclidean distance between their feature vectors, either for the entire image or for localized regions within the image. The operations involved in image search are frequently a nearest-neighbour search in Euclidean space, which *k*-d trees can accelerate the search step.

6.4 The Spatial Median *k*-d tree structure

Generally, building a *k*-d tree largely depends on where to place the splitting plane, as well as when to stop the recursion. The spatial median algorithm splits along a dimension p_k, chosen in a round-robin fashion, with the hyperplane positioned at the spatial median of the selected dimension. Considering a set of *k*-dimensional keys in a linear space $\mathbb{R}^n$, where a point in $\mathbb{R}^k$ is $(x_1, x_2, \ldots, x_k)$, each node will have a key and an associated split axis. The split axis i is an integer such that $0 \leq i \leq k-1$. The space is divided into two partitions with respect to $x[i]$. All keys y with $y[i] < x[i]$ go into the left partition/subtree, and all the keys with $y[i] \geq x[i]$ go into the right subtree. This is recursively applied to all subtrees, until empty nodes are reached. The splitting plane at each node is determined by alternating dimensions at each level, starting sequentially with $0, 1, 2, \ldots, k-1$.

Definition 1. *Generally, a k-d tree for a set of k-dimensional points is a binary search tree characterized by the following properties:*

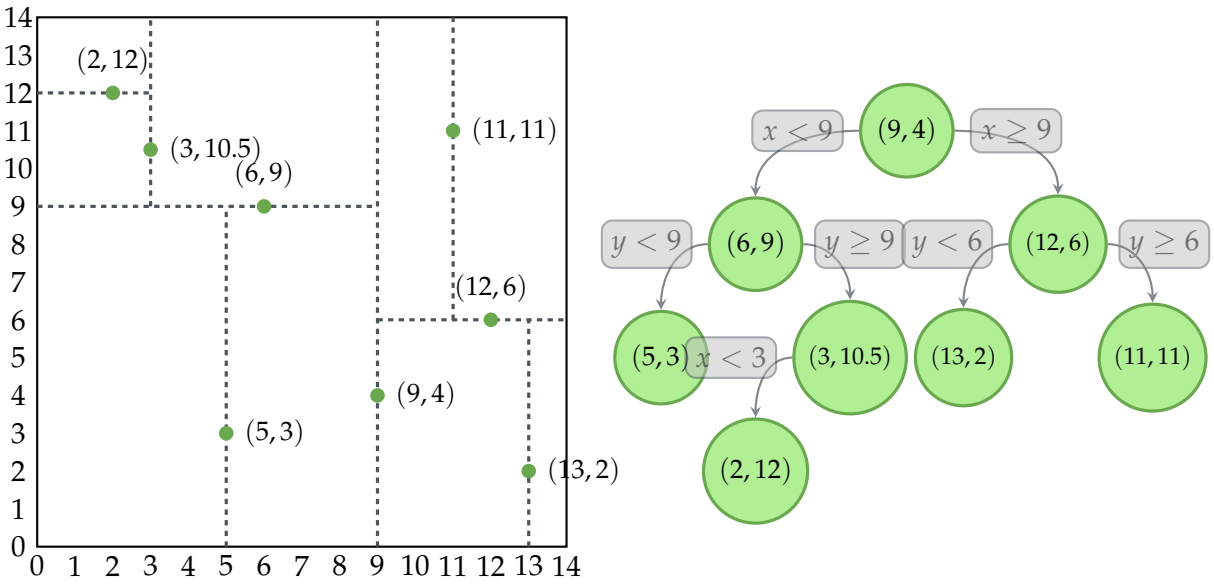
1. *Each node has a k-dimensional point and an associated split axis* $i \in \{0, \ldots, k-1\}$
2. *For every node with point x and split axis i, any point y in its left subtree satisfies* $y[i] < x[i]$*, while a point y in the right subtree satisfies* $y[i] \geq x[i]$.

3. *The root node has depth* 0 *and a split axis* 0. *A node at depth d has a split axis given by d* mod *k*.

Given a sample dataset $S \subseteq \mathbb{R}^n$ and split axis i, a canonical bulk build of the k-d tree based on this dataset will proceed making one node per loop with a recursive algorithm as follows:

1. Order the data points in S based on the i-th coordinates;
2. Choose the median point after the reordering as the value of the current node, and note the split axis i;
3. Set S_L as points before the median and S_R as points after the median point in S. S_L will be the left subtree and S_R will be the right subtree of the current node.
4. Reset $i \leftarrow (i + 1) mod\ n$. This is repeated 1 *to* 3 recursively on the left and right subtrees, alternating the dimensions at each level. The splitting process is applied sequentially across the n dimensions, with the procedure returning to the first dimension after the final dimension is reached.

For example, creating a k-d tree from the elements (9,4), (6, 9), (12,6), (5,3), (3, 10.5), (2, 12), (11, 11), (13,2) in a 2D space results in the following:



The cost T_c of constructing this median-split k-d tree with N items involves $\mathcal{O}(N)$ operations to sort T into T_L and T_R, in addition to the cost of recursively constructing the two children, $2T(\frac{N}{2})$, consequently:

$$T_c = N + 2T(\frac{N}{2}) = \cdots = \sum_{i=1}^{\log N} 2^i(\frac{N}{2^i}) = N \log N \tag{6.1}$$

6.5 Parallel *k*-d tree Algorithm

6.5.1 Tree Construction

Applications that employ the k-d tree can leverage on the data structure for fast traversals and high performance. However, the construction of the tree must be accelerated in order for applications to benefit from any performance gains. Given the task of building a k-d tree from P array data points in k dimensional space using T threads in a distributed shared memory system. The study assumes that both P and T are powers of 2 for ease of presentation, with $P \geq T^2$. This study adopts node-level parallelism to build the subtrees under each child independently. The recursive tree construction algorithm using the median-split approach is illustrated in Algorithm 6.1.

6.5.2 Tree Construction Optimisations

The section presents a distributed shared-memory implementation of the median-split k-d tree construction algorithm which leverages a novel data partitioning and distribution scheme. The approach provides optimised data affinity while ensuring that each thread works on a contiguous block of data to minimise remote accesses. The tree nodes created by each thread are kept within the shared space with data affinity to the respective thread.

Block-based data partitioning approach is adopted, where each thread is assigned contiguous blocks of data. The *upc_memget* and *upc_memput* functions are used to batch memory operations, thus minimizing the overhead of accessing remote data and improving overall cache locality. Consequently, the k-NN search minimizes remote memory accesses by ensuring that data-intensive operations are handled locally within each

thread's affinity region. The implementation optimises the tree construction using a thread-local approach, in which each thread constructs their *k*-d tree locally from assigned data blocks, allowing the threads to work independently. This approach further reduces global synchronization since threads can work independently on their assigned blocks. Additionally, data points are stored in a cache-friendly manner, such as using a struct of arrays (SoA) instead of an array of structs (AoS), and prefetch points and tree nodes where possible to minimize memory access latency.

Algorithm 6.1: Parallel *k*-d tree construction with Median-Split

Input: `Data points P` : $\{p_1, p_2, \ldots, p_n\}$, `start`: Starting index, `end`: Ending index, `depth`: Current tree depth

Output: Pointer to the root of the shared *k*-d tree

$axis \leftarrow depth \mod k$;

```
Sort points along the splitting axis;
quicksort_points(points, start, end, axis);
mid ← start + (end - start) / 2;

Allocate a node in shared memory;
node ← allocate_shared_node();

Copy the median point data to the node;
for d ← 0 to k - 1 do
    node->point[d] ← points[mid * k + d];
end

Recursively build the left and right subtrees;
node->left ← build_shared_kdtree(points, start, mid - 1, depth + 1);
node->right ← build_shared_kdtree(points, mid + 1, end, depth + 1);
return node;
```

6.6 *k*-d tree Search Algorithms

6.6.1 Orthogonal Range Search Algorithm

The orthogonal range search operation identifies all points contained within a defined (hyper)rectangle, specified by its lower and upper corners, $lowerBound$ and $upperBound$. The search returns all points x in the *k*-d tree bounded by the rectangle, such that $lowerBound[i] \leq x[i] \leq upperBound[i]$ for all i such that $0 \leq i < k$.

Given a set P of n data points in k-dimensional space $\mathbb{R}^k$, and an orthogonal range $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_1^{(Q)}, u_1^{(Q)}\right] \times \cdots \times \left[l_{k-1}^{(Q)}, u_{k-1}^{(Q)}\right]$, the task of answering queries about the subset of P that lies within the range Q is known as orthogonal range search. The information gathered about $P \cap Q$ varies depending on the query. The simplest queries include the reporting query, which lists all points in $P \cap Q$, and the counting query, which outputs the total number of points, $|P \cap Q|$. A standard orthogonal range search algorithm is shown in Algorithm 6.2 for completeness.

Orthogonal range search is commonly applied in database querying tasks, Geographic information systems, computer-aided designs (CAD), and computer graphics. In many of these applications, multiple queries are often performed on the same point set P , making it advantageous to leverage data locality and re-use.

6.6.2 Radius Range Search Algorithm

"Distance" numerically describes how far apart two points are in space, with Euclidean distance being one of the most widely used distance functions. Generally, given points p and q in n-dimensional space such that: $p = p_1, p_2, p_3, \ldots p_n$, and $q = q_1, q_2, q_3, \ldots q_n$. the Euclidean distance is given by:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}.$$

The distance function is required to meet the following criteria:

- Positive: $d(x, y) \leq 0$;
- The distance of a point to itself is zero: $d(x, y) = 0$ if and only if $x = y$;
- It is symmetric: $d(x, y) = d(y, x)$;
- It adheres to the triangle inequality property: $d(x, y) + d(y, z) \leq d(x, z)$.

Algorithm 6.2: Orthogonal Range Search in a k-d tree

Input: Root node of k-d tree *root*, query range $[L, U]$, dimension d

Output: Array of points within the query range

Initialize an empty list *result*;

if *root is a leaf node* **then**

  foreach *point p in the leaf node* **do**

    if $L[d] \leq p[d] \leq U[d]$ **then**

      Add p to *result*;

    end

  end

  return *result;*

end

if $L[d] \leq root.median$ **then**

  Add points from the left subtree satisfying $L[d] \leq p[d]$ to *result*;

  if $L[d] \leq root.median$ **then**

    Add points from the right subtree satisfying $U[d] \geq p[d]$ to *result*;

  end

end

else

  Add points from the right subtree satisfying $U[d] \geq p[d]$ to *result*;

  if $L[d] \leq root.median$ **then**

    Add points from the left subtree satisfying $L[d] \leq p[d]$ to *result*;

  end

end

Recursively search left and right subtrees in dimension $(d + 1) \mod k$;

return *result;*

6.6.3 Nearest Neighbour Algorithm

The nearest neighbour search algorithm returns the nearest points to a given point using a distance function. The k-nearest neighbour (k-NN) algorithm is a fundamental technique in machine learning and pattern recognition, and has been the focus of significant attention and research over the years. They are used for tasks involving classification and regression, enabling insightful predictions and patterns to emerge from data. In these tasks, they operate by identifying the k closest data points to a query

point, based on a specified distance metric, and making predictions based on the properties of these neighbours. For high-dimensional data, the *k*-NN algorithm can be computationally intensive because it requires calculating distances between the query point and all points in the dataset. The *k*-d tree is used to efficiently organize and search the data. By leveraging the hierarchical structure of *k*-d trees, *k*-NN search becomes more efficient, enabling a faster and scalable solution.

Using a *k*-d tree for k-nearest neighbour search involves navigating the tree structure to identify the *k* closest points to a specified query point. Generally, a *k*-NN search using a *k*-d tree proceeds as follows:

1. Starting with the root node, traverse the tree recursively to find the possible leaf node containing the query point. Traversal involves comparing the query point with the splitting hyperplane at each node, which determines movement to the left or right child node according to the relevant axis.

2. When a leaf node is reached, traversal backtracks to the parent node, where the distance between the query point and the corresponding hyper-rectangle is computed.

3. The process continues with Step 2 for each ancestor node, terminating at the root of the tree.

4. After visiting all relevant nodes, the priority queue will contain the *k*-nearest neighbours to the query point.

By leveraging the properties of the *k*-d tree, such as pruning branches that are unlikely to contain closer points, the *k*-NN search can be more efficient compared to brute-force methods, especially for large and high-dimensional datasets.

Algorithm 6.3: *k*-Nearest Neighbour Search using *k*-d tree

Input: Root node *root*, query point *query*, number of nearest neighbours *K*

Output: Priority queue *pq* containing *K* nearest neighbours

Initialize priority queue *pq* to store nearest neighbours;

$leaf_node \leftarrow$ `traverse_kd_tree(`*root*, *query*, 0`);`

current_node $\leftarrow$ $leaf_node$;

while *current_node is not null* **do**

    `backtrack(`*current_node*, *query*, 0`);`

    current_node $\leftarrow$ *current_node.parent*;

end

return *pq*;

Algorithm 6.4: Traverse *k*-d tree to find leaf node

Input: Node *node*, query point *query*, depth *depth*

Output: Leaf node containing the query point

if *node is a leaf* **then**
    return *node*

else

    $axis \leftarrow depth \mod k$;

    if $query[axis] < node.point[axis]$ **then**
        return `traverse_kd_tree(`$node.left$, *query*, $depth + 1$`)`

    else
        return `traverse_kd_tree(`$node.right$, *query*, $depth + 1$`)`

    end

end

Algorithm 6.5: Backtrack to add candidate nodes to the priority queue

Input: Node *node*, query point *query*, depth *depth*

if *node is null* **then**
    return;
end

$dist \leftarrow distance(node.point, query)$;

if $pq.size() < K$ ***or*** $dist < pq.max_distance()$ **then**
    $pq.insert(node.point, dist)$;
    if $pq.size() > K$ **then**
        $pq.remove_max()$;
    end
end

$axis \leftarrow depth \mod k$;

if $query[axis] < node.point[axis]$ **then**
    `backtrack`$(node.right, query, depth + 1)$;
end
else
    `backtrack`$(node.left, query, depth + 1)$;
end

if $pq.size() < K$ ***or*** $|query[axis] - node.point[axis]| < pq.max_distance()$ **then**
    if $query[axis] < node.point[axis]$ **then**
        `backtrack`$(node.left, query, depth + 1)$;
    end
    else
        `backtrack`$(node.right, query, depth + 1)$;
    end
end

Algorithm 6.6: Parallel k-NN Search

Input: k-d tree root node *node*, query point q, number of neighbors k
Output: List of k nearest neighbors

```
Function Parallelk-NN(node, q, k):
    globalQueue ← InitializePriorityQueue();
    Function 2:
        Search(node, q, k, queue)
    if node = ∅ then
        return
    axis ← depth of node mod k;
    if q[axis] < node.point[axis] then
        nearer ← node.left;
        farther ← node.right;
    else
        nearer ← node.right;
        farther ← node.left;
    Search(nearer, q, k, queue);
    UpdatePriorityQueue(queue, node.point, k);
    if |q[axis] − node.point[axis]| < distance to farthest in queue then
        Search(farther, q, k, queue);
    parallel for all processors;
    localQueue ← InitializePriorityQueue();
    Search(node, q, localQueue);
    MergeQueues(globalQueue, localQueue, k);
    end parallel;
    return ExtractNeighbors(globalQueue, k)
```

6.7 Performance Evaluation

6.7.1 Experimental setup

The study carried out a set of experiments to evaluate the performance of the distributed shared-memory implementation of the k-d tree algorithm. The performance of the augmented k-d tree implementation algorithm was compared with FLANN: a state-of-the-art library which contains a collection of optimised algorithms for performing fast approximate nearest neighbour searches in high-dimensional spaces. The performance evaluations were conducted using up to 256 CPU cores on the Lengau Cluster of the Centre for High Performance Computing, South Africa. The homogeneous cluster consists of Dell servers which are powered by 5th generation Intel processors with 56 Gbps FDR Infiniband interconnect. Each compute node consists of 24 2.6 GHz Intel

Xeon E5-2690 v3 processors with 64GB memory.

All codes were implemented in C and compiled with the Berkeley Unified Parallel C compiler version 2022.10.0 or MPICH 3.4.3. The experiments were carried out using real Geospatial Map Data of South Africa from the OpenStreetMap (OSM) powered repository, Geofabrik [143]. The OpenStreetMap data for the region consists of more than a million GIS points from different feature points in South Africa. In this experiment, the GeoJSON files were downloaded and converted to Comma-separated values (CSV) with longitudes and latitudes using the *osmium* command line tool [144], to allow for easy experimentation. The performance of the augmented *k*-d tree implementation was compared with the appropriate functions of the latest version of the FLANN library: version 1.9.2.

Each test was repeated three times, with the average execution time (difference between start and end times) reported. Execution time was measured using *MPI_Wtime()* for MPI and *upc_ticks_now()* with *UPC_TICKS_PER_SEC* for UPC, both providing high-precision wall-clock time in seconds to capture computation time.

6.7.2 Results And Discussion

The efficiency of the algorithm was evaluated through a series of experiments, benchmarked against the *k*-d tree index of the FLANN library. Performance evaluation encompassed both tree index construction and traversal operations, including nearest-neighbour and range queries, using up to $1,048,576$ data points in a 2D space from the GIS dataset. For each experiment, the tests were executed 3 times and the average latency was recorded. The goal of these tests was to see how distributing across several nodes would affect operational throughput, as well as evaluate the scaling efficiency of the algorithms with varying resources and workloads.

Building the *k*-d tree Index Structure

The initial data distribution times of the algorithms were profiled and compared. Figure 6.1 shows the data results, which highlights the differences in latencies for data

distribution operations with varying numbers of processors. As the number of processors increases, load times and associated latency generally increase due to increased communication among nodes. However, unlike the augmented UPC implementation, the FLANN index requires explicit message passing to distribute data chunks across the processors. The latency increases significantly when the number of cores is 256. The load times from the augmented implementation are relatively low (a maximum time under 6 seconds compared to a maximum of $\approx$ 22 seconds for FLANN), leveraging on the underlying high-performing GASNet layer in its communication. The nodes are assigned their requisite block chunks with affinity in the shared space. For clarity in the plots, the augmented *k*-d tree algorithm is denoted as "KDT".

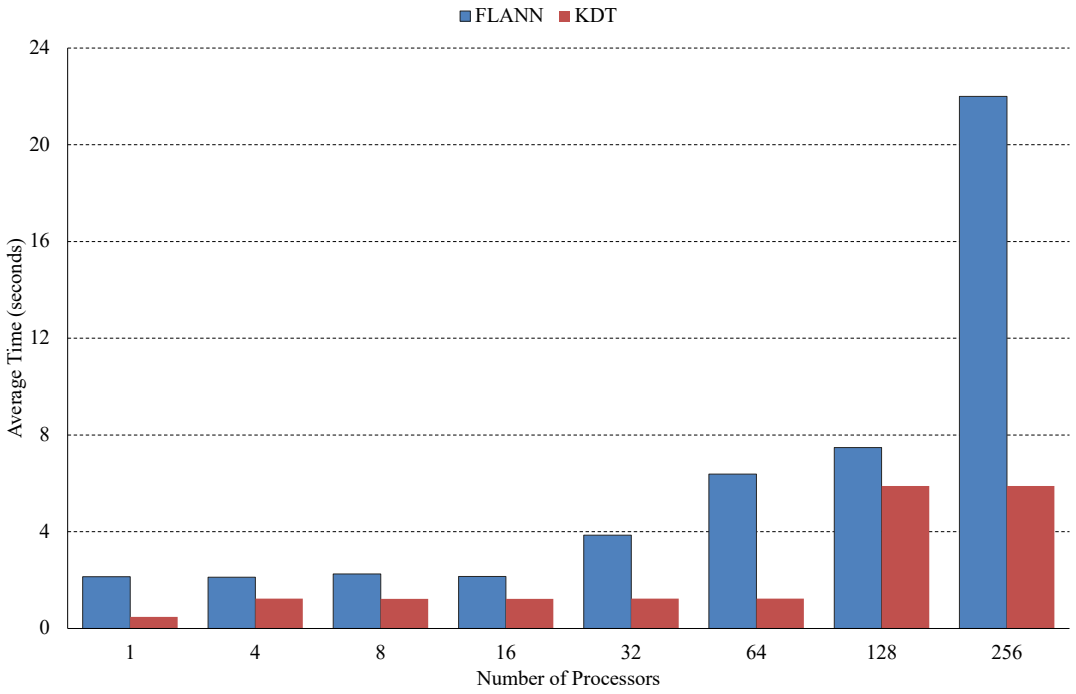


FIGURE 6.1: Increasing timing for Initial data distribution across processors

Figure 6.2 presents the average construction times to build the tree index using up to 1,048,576 data points (in a 2D space) from the real word GIS dataset. FLANN-256, 128, 64 and 32 represent FLANN results using 256, 128, 64 and 32 processor cores respectively, while KDT-256, 128, 64 and 32 represent results from the augmented algorithm using 256, 128, 64 and 32 processor cores respectively. The figure shows higher build times for relatively small-size datasets: for instance, FLANN-256 with 4096 data points takes over 1.6 seconds, whereas KDT-256 takes 0.3 seconds. However, FLANN-256 with

1,048,576 data points takes 1.2 seconds, whereas KDT-256 takes under 0.2 seconds. This could be due to increased communication overhead, particularly for relatively higher number of processor cores and smaller datasets. Index construction times were found to increase with larger dataset sizes. As the size of the multidimensional dataset increases, more work is required from each processor since the block size per core also increases, resulting in more computation and a corresponding increase in latencies.

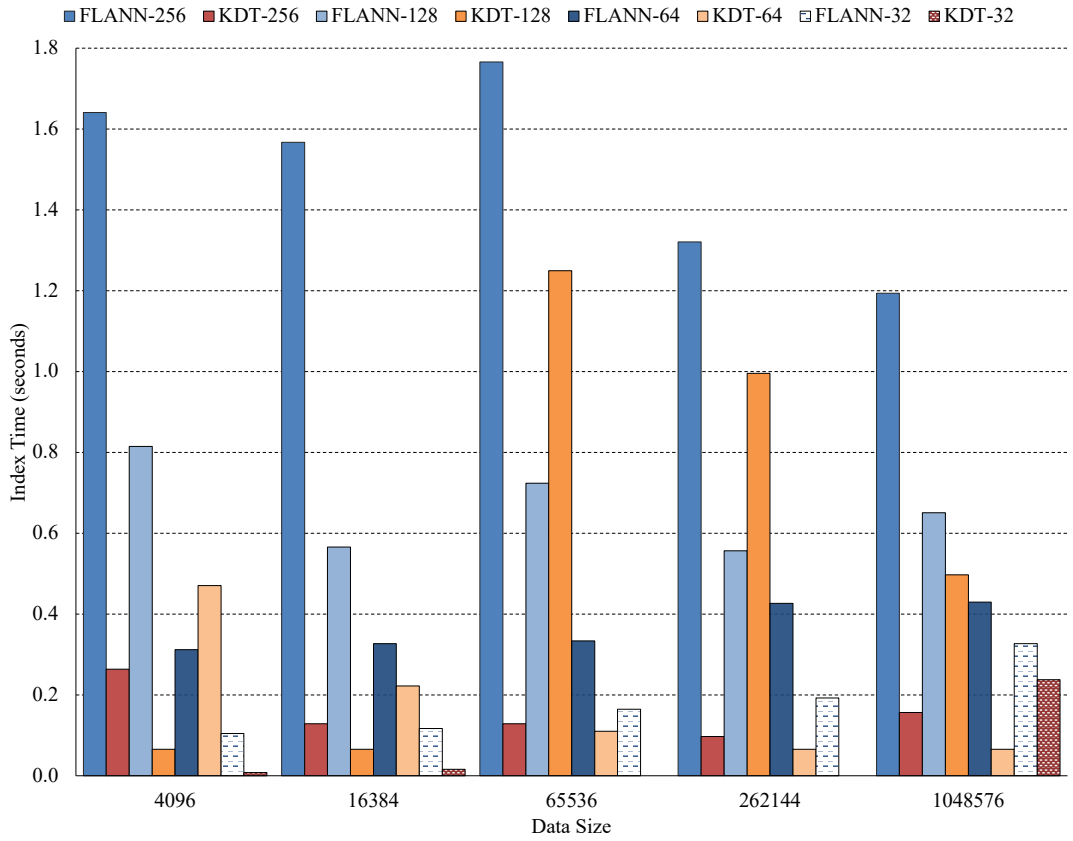


FIGURE 6.2: *k*-d tree Index Build time for varying workloads and cores

This is demonstrated in Figure 6.3 where the data size is fixed at 1,048,576 while increasing the number of processor cores from 1 to 256. The latencies generally decrease with increasing cores, since the data partitioning ensures that each thread receives a relatively smaller chunk as cores increase. For example, for 256 processor cores, and 1,048,576 data points, each processor will have 4096 data points to process. This results in relatively less workload required by each thread/core. However, latency begins to rise beyond 32 cores for FLANN and beyond 64 cores for KDT. At this point, communication and synchronization overheads seem to outweigh parallelism with increasing cores. The KDT algorithm demonstrated performance comparable to that of

the FLANN library and performs up to over 6× better as the number of cores increases. The thread-local approach ensures less inter-node communication, and ultimately minimised synchronisation, while leveraging on data locality to ensure high performance and reduced latencies. This naturally translates into relatively higher throughput. In general, more cores will result in fewer data blocks per core and, therefore, faster tree construction times and effectively lower latencies.

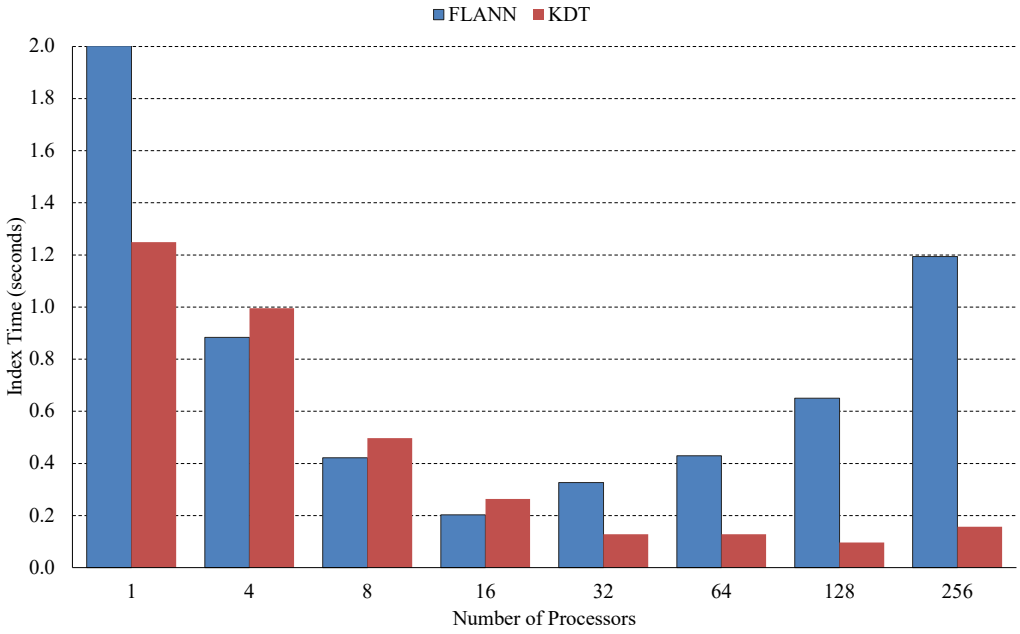
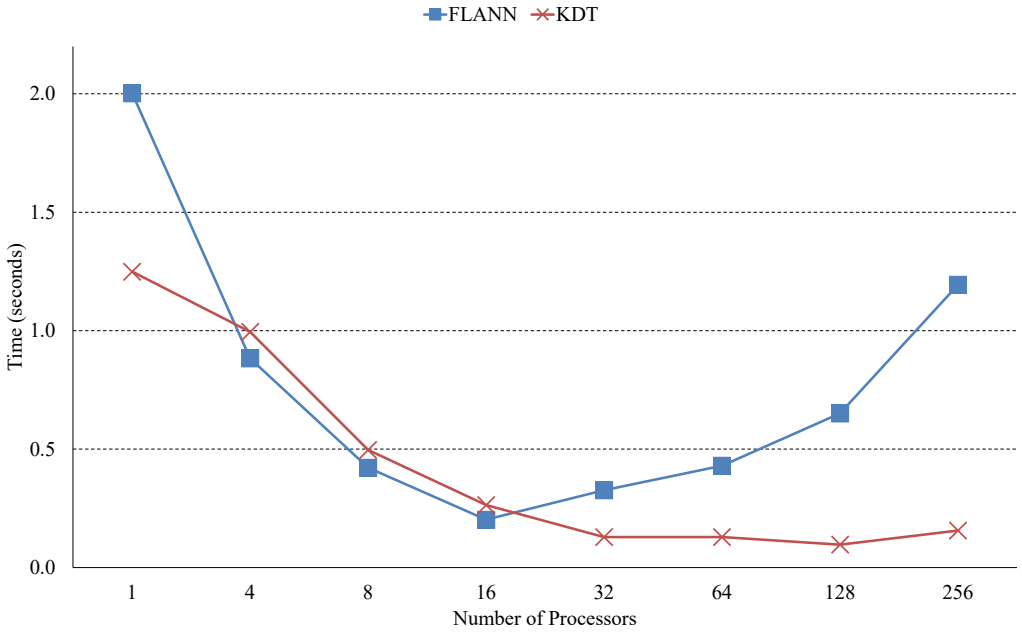


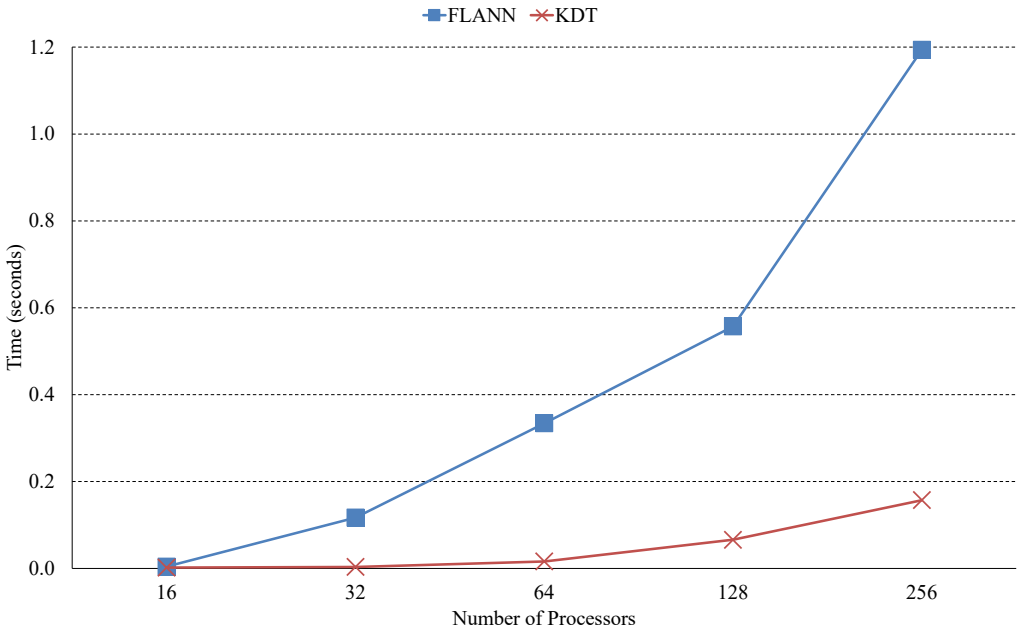
FIGURE 6.3: *k*-d tree Index Build time with 1,048,576 GIS dataset for varying and cores

The results of the scaling evaluations are presented in Figures 6.4a and 6.4b. In the strong scaling test, a fixed workload of 1048576 was maintained while increasing the number of processing cores for each algorithm up to 256 cores. Tree index construction was performed three times (3×), and the average build time was recorded.

Generally, in the strong scale test, as the number of processor cores/threads increases, the workload for each core reduces, and therefore the build time reduces, and the operational throughput for that matter increases. For FLANN, the latency begins to increase again after 16 cores, whereas KDT experiences a minor increase after 32 cores. At these points, the communication overhead increases the overall latency, reducing the algorithmic performance. Generally, as more computing resources are made available, it



(A) Strong scale



(B) Weak scale

FIGURE 6.4: Evaluation of *k*-d tree construction scaling efficiency. For strong scaling, the workload size was fixed at 1048576. For weak scaling, the workload was increased from 4096 to 1048576, while the number of processors was doubled from 16 to 256.
Note: lower is better. Discrete points are connected to illustrate performance trends as processor count increases.

is expected to have increase parallel efficiency/increase speed-up given less interprocess communication. The thread-local index construction approach ensures minimised interprocess communication, resulting in good scaling efficiency for KDT. Figure 6.5

shows the strong scaling speed-up plots for the two algorithms, highlighting KDT's superior scalability with increasing number of processor cores.

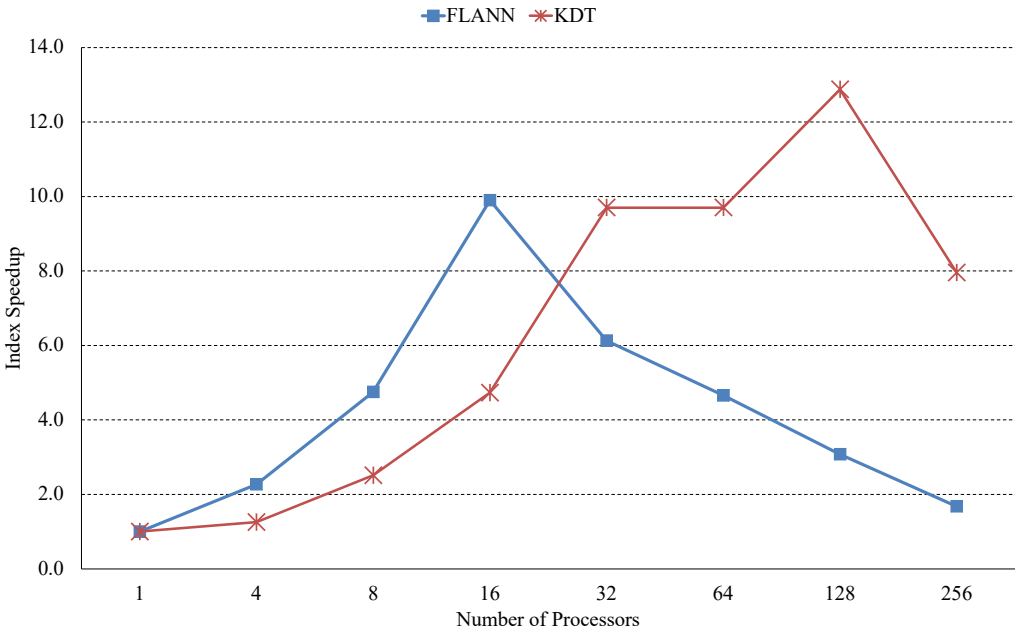


FIGURE 6.5: *k*-d tree Index Build Speed-up

The weak scaling test (Figure 6.4b) shows a general increase in latency as the cores and the size of the workloads are doubled. The results show that the KDT algorithm exhibits a slower rate of latency increase compared to the FLANN tree index. Although block sizes increase with workload, the reduced synchronization ensures that KDT incurs less latency and ultimately achieves a better performance. Figure 6.6 showcases the weak scaling efficiency plots for the two algorithms, highlighting the superior scalability of KDT with an increasing number of processor cores, particularly after 32 cores. The weak scaling efficiency is given by $T(1)/T(N_p)$ where: $T(1)$ is the time to construct the index using 1 processor (serial task), and $T(N_p)$ is the build time for the same data size with N_p processors (parallel task).

Search Operations on the *k*-d tree Index Structure

Further investigation considered the use of the constructed tree index for search operations, such as nearest-neighbour queries in a 2D space with randomly selected data points. The *k*-NN algorithm is widely used in both Geographic Information Systems (GIS) and machine learning for spatial analysis and classification tasks. In the GIS

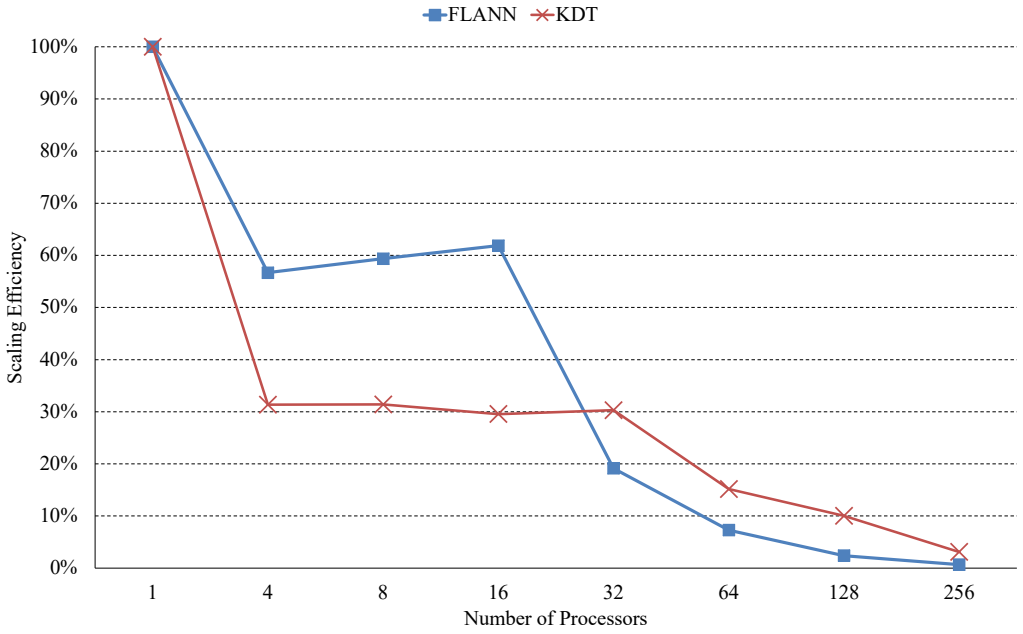


FIGURE 6.6: *k*-d tree Index Build Scaling efficiency

dataset, *k*-NN is particularly valuable for finding the closest geographic locations, such as identifying the closest hospitals, schools, or emergency response centres relative to a specific query point. Using the *k*-d tree index, the *k*-NN algorithm can efficiently handle such large geographic datasets to perform proximity searches with minimal computational overhead. In this test, the query point is set to the coordinates [-26.2041, 28.0473]: a location within the Johannesburg downtown area, and requested the 5 closest points ($k = 5$) based on the available dataset.

Figure 6.7 presents the results of the nearest-neighbour search of the tree index with varying data workloads. The search time generally increases as the size of the tree index increases, since the search has to traverse more nodes to identify probable neighbours. Each thread independently traverses their local tree index and updates the global queue with their results. This improves data locality and minimises global synchronisations, resulting in relatively lower latencies and resulting higher operational throughput and efficiency. Figure 6.8 shows the relative superior performance of KDT for a larger number of processing cores.

The results of the scaling tests of the *k*-NN algorithms using the *k*-d tree spatial index structures are presented in Figures 6.9a and 6.9b The scaling efficiency test assesses the spatial index structure's capacity to efficiently handle varying workloads and processor

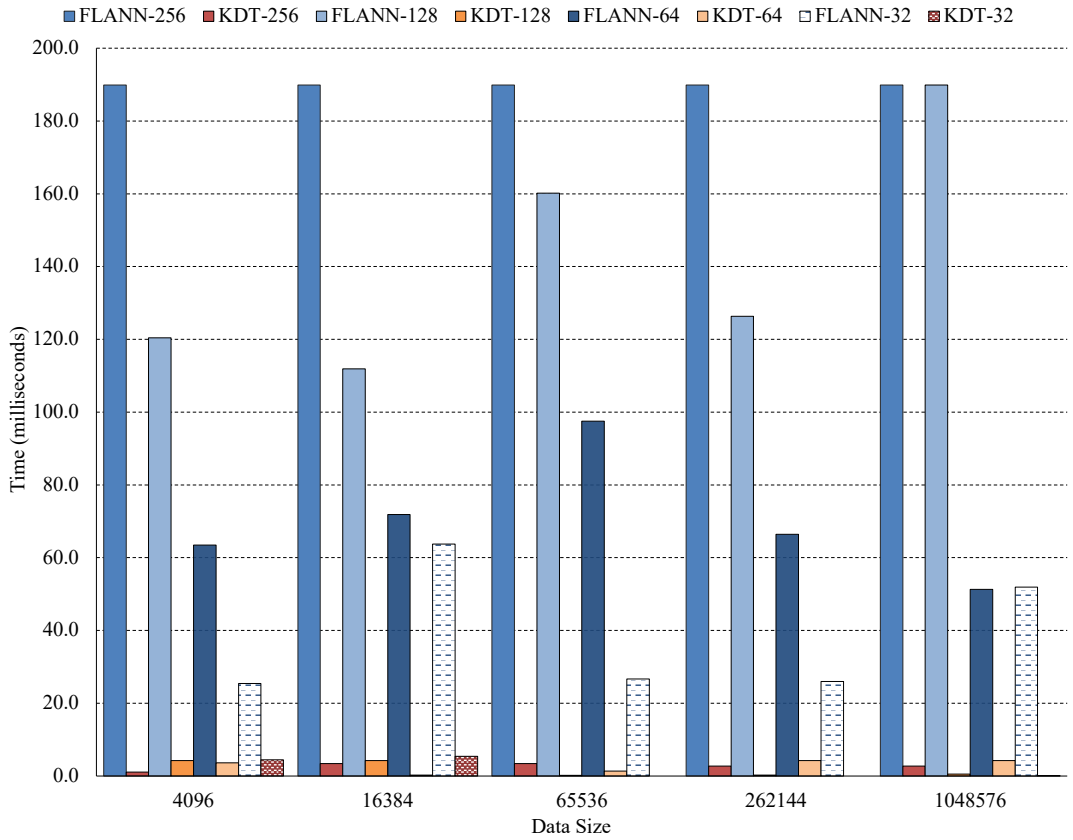


FIGURE 6.7: *k*-NN Search time for varying workloads and cores

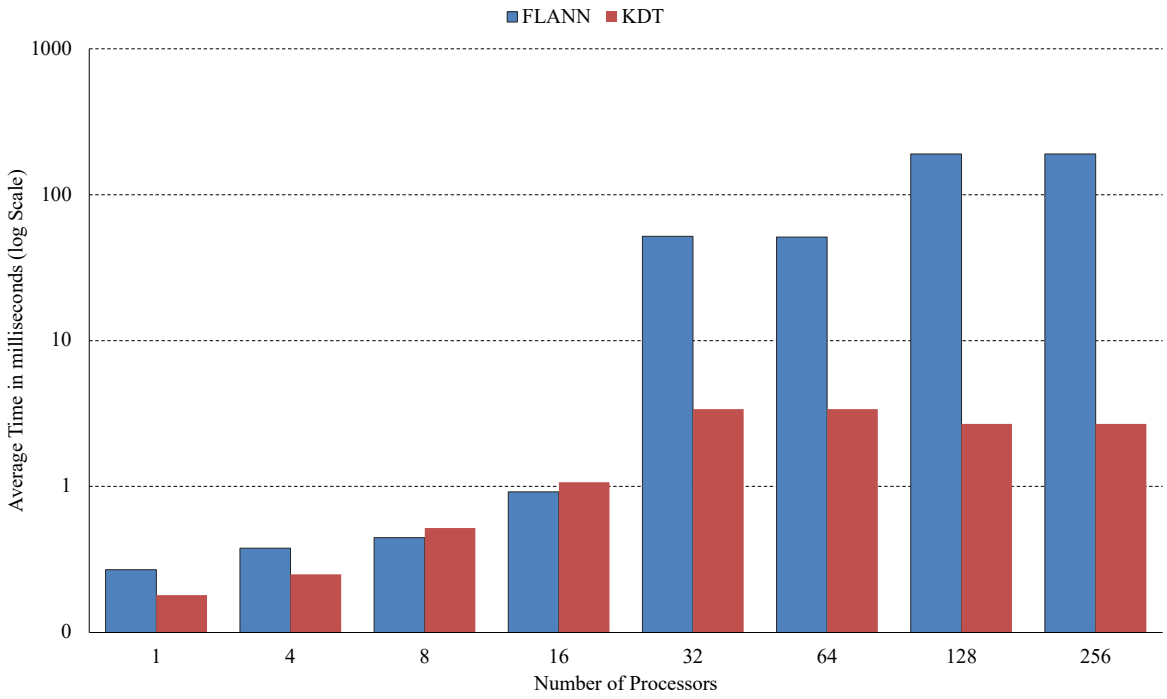
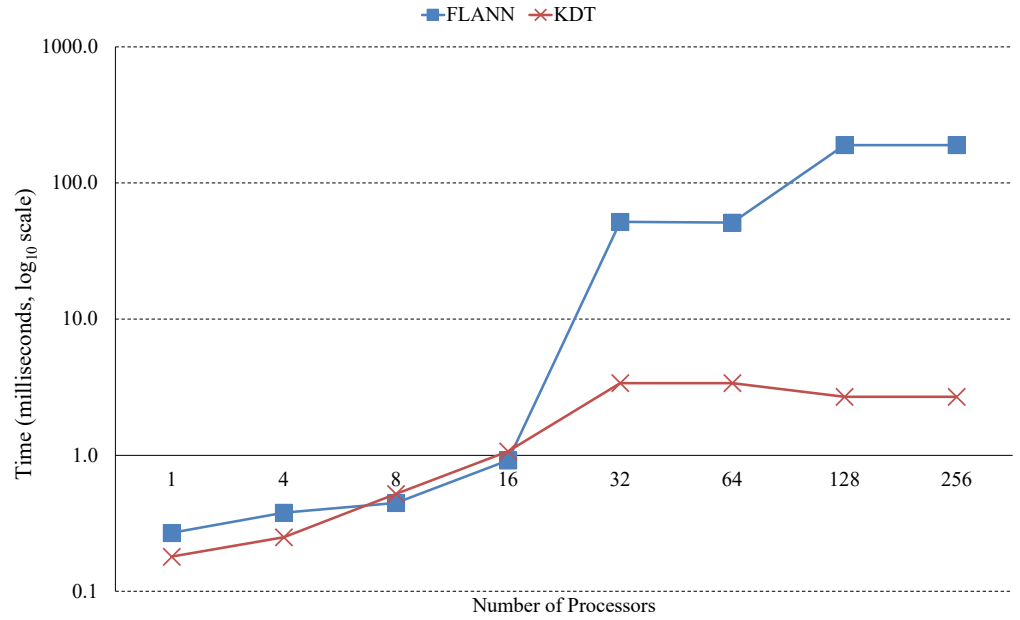
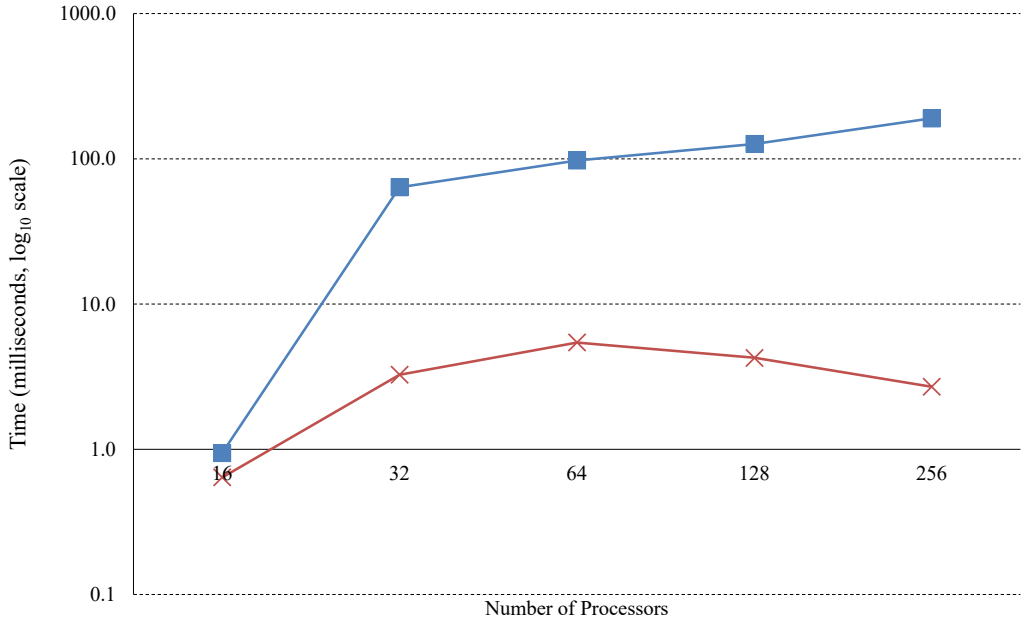


FIGURE 6.8: *k*-NN Search time using 1,048,576 dataset for varying and cores



(A) Strong scale



(B) Weak scale

FIGURE 6.9: Evaluation of *k*-NN scaling efficiency between FLANN & KDT in $\log_{10}$ scale. In the strong scaling test, a fixed size workload of 1048576 was maintained. In the weak scaling test, the workload was doubled from 4096 to 1048576, while the number of processors was simultaneously doubled from 16 to 256.
Note: lower is better

cores/threads. In the strong scale test (Figure 6.9a), the worker threads are increased from 16 to 256 while the tree index is fixed with 1048576 data points. As the number of threads is increased, less work (data points per thread) is required by each thread/core

and therefore the search time reduces while the operational throughput increases. The *k*-NN search throughput generally increases as the number of threads increases for a fixed-size tree index. In general, a well-balanced index tree structure ensures fast and efficient traversals, and eventually a fast search. Performance results show that the KDT implementation outperforms the FLANN, with gains of up to 1.5×. The search algorithm leverages the thread-local advantages to ensure that each thread proceeds independently.

Figure 6.9b shows the latencies for the weak scaling test in which the tree index size is increased from 4096 to 1048576 while increasing the processor cores from 16 to 256. The index size increases proportionally to the number of threads/processor cores. For example, when the tree index size is doubled, the number of threads is doubled, so that the compute load for each processor/thread remains the same with increasing processors. The KDT algorithm demonstrated better scaling efficiency compared to the FLANN library.

6.8 Chapter Summary

The *k*-d tree is a binary search tree for organizing points in a *k*-dimensional space. The data structure allows for efficient range and nearest-neighbour queries. It is a space partitioning data structure for organizing points in a *k*-dimensional space. They have been widely studied and used in many applications, including ray tracing and spatial indexing in Computer Graphics; motion planning and collision detection in robotics; spatial databases and mapping services in Geographic Information Systems; classification and clustering algorithms in machine learning; and many more. In machine learning, for instance, *k*-NN which is a key non-parametric algorithm used for classification, regression, and anomaly detection, leverages on the *k*-d tree index. It identifies the k-closest data points in a feature space based on a distance metric such as the Euclidean distance and makes predictions by aggregating the properties of these neighbours. Making tree operations like building and traversing better ensures better performance for these applications.

In this chapter, the study presented a *k*-d tree implementation for distributed shared-memory leveraging in-memory processing. The augmented algorithm ensures efficient data partitioning, allowing for thread-local index construction, including strategies to optimise search algorithms (like range, orthogonal and *k*-nearest neighbours (*k*-NN)) The parallel subtree construction, cache sensitive, and locality-aware data layout ensures a high-performing tree index construction and ultimately fast queries from the well-balanced tree index. The chapter presented performance evaluations using real-world GIS dataset, illustrating performance gains with the indicated optimisation techniques for the median split *k*-d tree compared to the state-of-the-art FLANN library. This study further shows that the KDT algorithm has superior performance for both index construction and searches: up to 1.5× for traversals, and 5× for index build.

Chapter 7

Conclusion And Future Direction

7.1 Summary of Research contributions

7.1.1 The Elastic PGAS concept And Performance Benchmarks

The shift towards many-core architectures increases the I/O-stack contention, which presents performance challenges for parallel scientific applications. The increasing generation, analysis, and management of multiple extreme-scale datasets, coupled with the advent of new storage and networking technologies, exert increasing pressure on the storage capabilities of current HPC systems. Modern HPC systems routinely handle large-scale scientific computations and simulations that generate large amounts of data for storage. The storage and I/O systems are under increasing pressure due to such big data. However, when disk I/O is minimised, applications achieve significant performance gains. I/O is commonly acknowledged as a crucial performance constraint for large-scale computation, and data analysis, and it is anticipated that this situation will exacerbate with the continual exponential rise in the discrepancy between computation and I/O capacity, particularly on emerging exascale systems. By minimizing I/O and utilizing distributed memory storage during computation with the EPGASS strategy, a significant improvement in performance gains can be achieved.

Chapter 3 presents the basic principle behind the EPGASS architecture using commodity hardware and systems leveraging in-memory computing (IMC) to provide high performance I/O instead of specialised and vendor-locked systems which could be difficult to manage, and expensive to maintain in the long term. The architecture achieves

high performance by leveraging the partitioned global address space paradigm to ensure significant use of memory resident data. The concept proposes dedicated compute nodes configured with large DRAMs or NVRAMs, as well as SSDs; Storage I/O nodes enhanced with a hierarchical storage stack including a parallel file system and a high bandwidth interconnect between nodes to provide performance enhancement, particularly, for applications with large memory footprint. During the application run, checkpoint data is stored in memory and asynchronously persisted through a hierarchical storage structure, which includes high-speed memory, fast SSD, and subsequently, relatively slower storage disks. For many applications such as image processing, GIS data processing, and many more, which have large memory footprint, leveraging EPGASS provides an opportunity to achieve high throughput by minimizing I/O usage during active computation.

In Chapter 4, the study evaluated the throughput performance of various benchmarks on the EPGASS setup, comparing both UPC and MPI implementations, as well as their memory efficiency. The NAS Parallel Benchmarks experiments with both implementations demonstrated a fairly comparable performance, but some performance variances due to the underlying differences in communication protocols. The results show the performance gains using the distributed shared memory paradigm. As a distributed shared-memory model, UPC performed better consistently, showing higher throughput compared to the MPI implementations. The results demonstrated that the partitioned global address space allows for a relatively more efficient computation on the proposed setup. The performance enhancements could be attributed to the effective management of the distributed shared memory, the reduced communication overhead resulting from the effective data distribution, and the efficient synchronization mechanisms. When it comes to communication-intensive tasks, efficient data handling and reduced communication latency, provide substantial performance gains. It is noteworthy that the underlying high-performance GASNet communication layer facilitates superior performance and scalability, including remote memory accesses. This study further investigated the memory usage of each of the different models, where a relatively comparable memory requirements across the different tests was observed, with MPI requiring less memory in various instances compared to UPC.

Furthermore, the STREAM benchmark was used to evaluate memory utilization for both strided and random accesses, which represent common data access models in scientific applications. Utilization was found to decrease as the number of threads increased for both access methods. The increase in the number of threads results in relatively lower memory usage across different threads. We, however, noted strided accesses performed much better as expected, pointing to the need for efficient data organisation, and optimizations for various distributed parallel processing tasks. The results further highlight the inherent challenges and overheads associated with different memory access patterns, noting that decreasing memory bandwidth utilization with increasing processors is primarily due to increased contention and overhead costs in accessing shared resources. Using efficient data partitioning and algorithms that require less synchronization while leveraging local data can help parallel applications run faster and use memory better. This optimization is essential to achieve efficient and scalable parallel data processing, particularly in applications like matrix-matrix and matrix-vector multiplications, which could involve several data movements and synchronisation. An efficient data partitioning technique to minimise data movement and synchronisation is therefore essential to ensure high-performance and scaling efficiency.

7.1.2 The Single Matrix Block Shift (SMBS) algorithm And EPGASS

Many algorithms for scientific and numerical computations depend on matrix-matrix or matrix-vector multiplication as a basic operation. These serve as the foundation for numerous matrix operations used in numerical solvers and graph-theory problems. Various algorithms, including Cannon, PUMMA, SUMMA, and SRUMMA, have been proposed for implementing matrix-matrix multiplication, particularly on distributed architectures. In distributed architectures, large matrices are decomposed into smaller chunks or blocks and distributed across a grid of processors to enable parallel processing of the solution, allowing for high performance efficiency in computing the matrix solution. However, this involves several communication and synchronisation, as the grid of processors need to communicate and distribute the matrix blocks during the computation.

Communication overhead is well recognized as accounting for more than half of the total latency in many large-scale distributed algorithms, including matrix-matrix and matrix-vector multiplication. In Chapter 5, the study proposes a novel distributed matrix-matrix multiplication algorithm, the Single-Matrix-Block Shift (SMBS) algorithm, which minimises the communication overhead costs, and scales efficiently with increasing number of computing cores. The algorithm leverages on the memory efficiency of the Cannon algorithm, and improves upon the algorithmic performance by limiting the amount of communication for dense matrix-matrix multiplication in distributed systems. The analytical results presented demonstrate performance improvements over the Cannon algorithm and establish theoretical bounds comparable to those of the state-of-the-art SRUMMA algorithm. The theoretical upper bound for the Cannon algorithm is $\mathcal{O}\left(\frac{N^3}{P}\right) + \mathcal{O}\left(\frac{N^2}{\sqrt{P}}\right)$, with the performance improving as P increases until the communication overhead becomes a limiting factor. SMBS, on the other hand, has a theoretical upper bound of $\mathcal{O}\left(\frac{N^3}{P}\right) - \mathcal{O}\left(\frac{N^3}{P^{\frac{3}{2}}}\right) + \mathcal{O}\left(\sqrt{P}\right)$, while SRUMMA has latencies bounded by $\mathcal{O}\left(\frac{N^3}{P}\right) + \mathcal{O}\left(\frac{N^2}{\sqrt{P}}\right) + \mathcal{O}\left(\sqrt{P}\right)$.

The experimental evaluations demonstrate the superior performance of SMBS compared to the Cannon algorithm with throughput comparable to a tuned state-of-the-art SRUMMA algorithm for some workloads. SMBS achieves higher performance by limiting the number of block shifts, as well as block skews, and consequently performs relatively fewer data movements (hence reducing the amount of communication) compared to the Cannon algorithm. Additionally, unlike Cannon and other well-known algorithms (such as SRUMMA), the SMBS algorithm provides the final block results during each computational loop. Each iteration of the augmented algorithm produces $2 \times \mathit{BlockSize}$ of the final matrix C. Thus, eliminating the last reduction step, since the final results become available per iteration. This provides an elegant approach for image processing applications, in particular, which may require some parts of the final output while the application processing is in progress. Consequently, minimising data movement results in exploiting data locality, and further enhances the algorithmic performance. Matrix multiplication applications can benefit from the performance optimizations introduced by SMBS, achieving further performance improvements and

increased processing throughput. This is especially true for non-blocking implementations, which utilize non-blocking point-to-point messaging to overlap communication with computation. The performance evaluations indicate throughputs over $5X$ compared to the blocking SMBS for dense matrix-matrix multiplication.

Furthermore, this study presented the implementation of SMBS employing the PGAS distributed shared-memory approach and evaluated the concept using both synthetic and real-world datasets for image processing and image classification. The results show that exploiting data locality and cache coherency further results in improved algorithmic performance. In addition, the use of real-world image processing dataset, CIFAR-100, illustrated the scaling efficiency of the algorithm in real-world scenarios for image dimensionality reduction and feature extraction (which are part of the convolutional neural network pipeline). The implementation of distributed shared memory reduces the complexity of direct message passing, providing a shared memory storage that can be accessed by all processors, thereby significantly reducing the overhead latencies associated with setting up and managing communication channels.

7.1.3 *k*-d Trees And EPGASS

The *k*-d tree is a binary search tree for organizing points in a *k*-dimensional space. It is a space partitioning data structure for organizing points in a *k*-dimensional space, allowing for efficient range and nearest neighbour searches. They have been widely studied and used in many applications such as ray tracing and spatial indexing in computer graphics; motion planning and collision detection in robotics; spatial databases and mapping services in Geographic Information Systems; classification and clustering algorithms in machine learning, and many more. Optimising performance of the various tree operations such as construction and traversals results in better performance for these applications.

Chapter 6 presented a *k*-d tree implementation for shared memory distributed systems, including strategies to optimise both tree construction and search algorithms (like orthogonal and k-nearest neighbours) leveraging on the EPGASS concept, and parallel processing capabilities such as parallel subtree construction, cache sensitive

and locality aware data layout. The chapter presented performance evaluations illustrating performance gains with the proposed optimisations techniques for the median-split *k*-d tree, including evaluations using real-world GIS dataset for nearest-neighbour queries. The proposed algorithm is evaluated against the state-of-the-art FLANN library. The FLANN library is a highly efficient and optimised open-source library designed for performing fast approximate nearest-neighbour queries, particularly in high-dimensional space. FLANN's *k*-d tree implementation supports both single and multiple *k*-d trees, where multiple *k*-d trees improve query performance for the approximate queries. FLANN is especially suitable for large-scale applications in computer vision, robotics, and machine learning where *k*-NN searches are often needed. Unlike, the FLANN library, the augmented KDT algorithm leverages optimised data layout, and thread-local approach to enhance algorithmic scalability and minimise global synchronisations while ensuring a well-balanced tree index to support queries. The experimental evaluations show a comparable performance of the KDT algorithm compared with the FLANN's *k*-d tree implementation both in building the tree index and search queries including nearest-neighbour queries.

7.2 Future Directions

The main idea behind the EPGASS concept is to optimise for performance by leveraging in-memory computing while taking advantage of hierarchical memory structure, and optimal cache sensitive data layout. Future work will explore benchmarking memory elasticity across larger node counts to evaluate elasticity overhead compared to memory efficiency during computation. In addition, further experiments are anticipated on leveraging the ADIOS2 API for in-memory data usage and integration with elastic memory management.

Besides, the *k*-d tree implementation and performance optimisations, directions for future work will explore the application usage of EPGASS for multidimensional indexing schemes such as R-Trees; the Smith–Waterman algorithm which is used for local sequence alignment to determine similar regions between two strings of nucleic acid sequences or protein sequences [145]. Performance improvements are expected as these

algorithms are optimized using the distributed shared-memory paradigm and efficient data distribution and data layout strategies.

Additionally, future exploration of EPGASS on GPU/GPGPU architectures is planned to leverage the parallel computing capabilities of these platforms. GPUs are designed for massively parallel computation and manage shared memory differently. The implementations on GPU/GPGPU architectures will require efficient management of the Streaming Multiprocessors (SM) as well as thread blocks. A GPU/GPGPU-based implementation of the Single Matrix Block Shift (SMBS) algorithm is particularly of interest, as it may offer opportunities for performance optimization tailored to such GPU architecture. Current implementations of the SMBS algorithm work best on dense matrix-matrix multiplication, where most of the matrix elements are not zeroes. Sparse matrices, on the other hand, have a higher proportion of zero elements. For the storage and manipulation of sparse matrices, it becomes imperative to employ specialized algorithms as well as data structures that leverage on the sparse structure of the matrix to optimise for performance gains. Future work of SMBS will explore possible optimisation techniques to adopt the algorithm for sparse matrices, while taking advantage of its data locality and performance gains.

Appendix A

Appendix

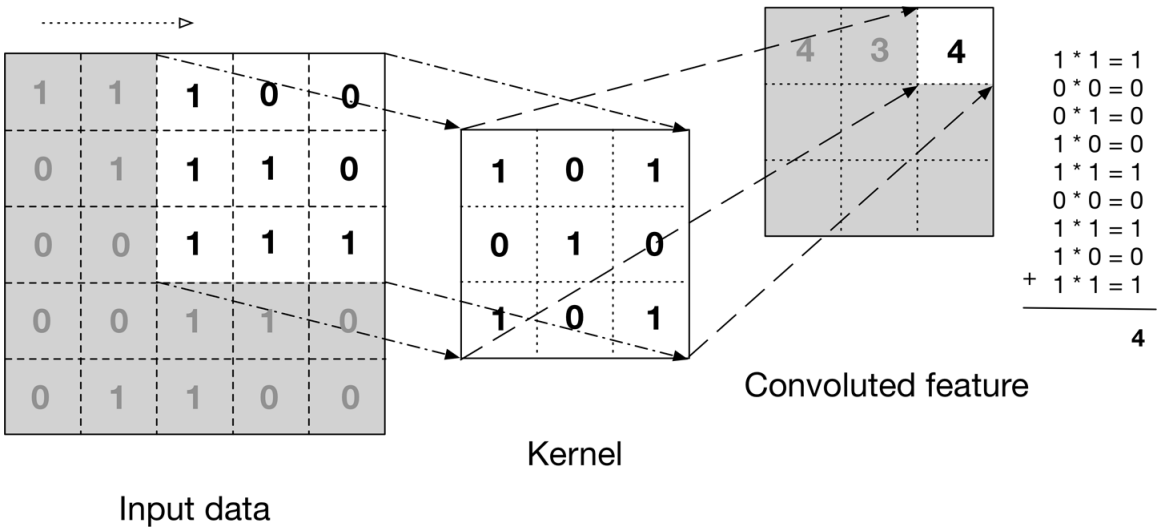


FIGURE A.1: An illustrative image showing the concept of convolution where, a 3×3 matrix kernel is applied to the input image to produce the convolved feature which is then forwarded to the next layer

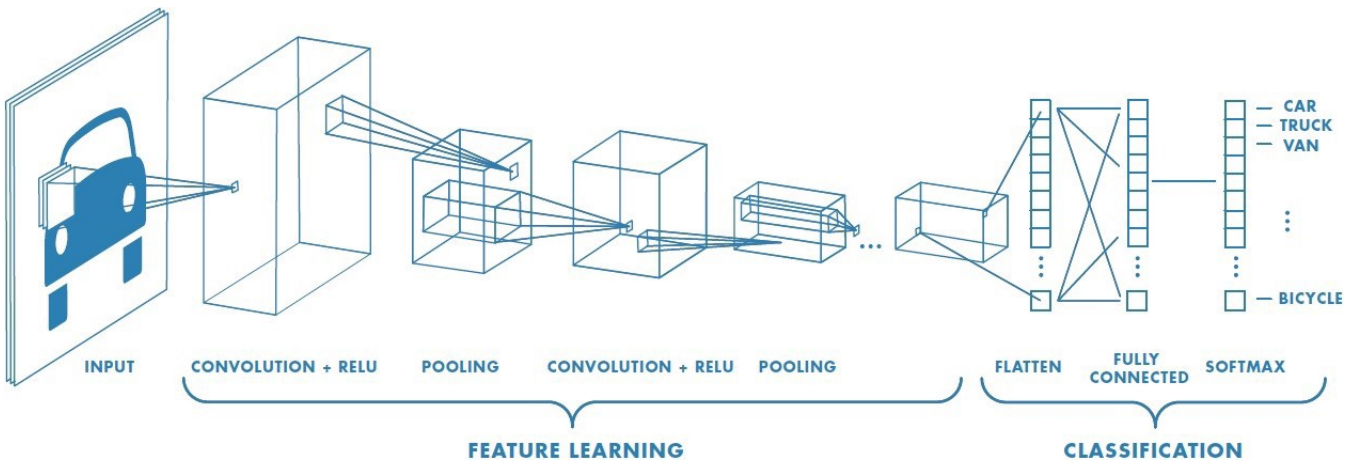


FIGURE A.2: Typical layout of the convolutional neural network highlighting feature extraction layer where kernel is applied
[1]

References

[1] Awais Khan, Paul Arnab K., Christopher Zimmer, Sarp Oral, Sajal Dash, Scott Atchley, and Feiyi Wang. Hvac: Removing I/O Bottleneck for Large-Scale Deep Learning Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 324–335, 2022. doi: 10.1109/CLUSTER51413.2022.00044.

[2] Lipeng Wan, Axel Huebl, Junmin Gu, Franz Poeschel, Ana Gainaru, Ruonan Wang, Jieyang Chen, Xin Liang, Dmitry Ganyushin, Todd Munson, Ian Foster, Jean-Luc Vay, Norbert Podhorszki, Kesheng Wu, and Scott Klasky. Improving I/O Performance for Exascale Applications Through Online Data Layout Reorganization. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):878–890, 2022. doi: 10.1109/TPDS.2021.3100784.

[3] Juan Fombellida, Irene Martín-Rubio, Santiago Torres-Alegre, and Diego Andina. Tackling business intelligence with Bio-inspired Deep learning. *Neural Computing and Applications*, 32:13195 – 13202, 2018.

[4] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam T. Moody, Kathryn M. Mohror, Kento Sato, and Weikuan Yu. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156, 2018.

[5] Cong Xu, Suparna Bhattacharya, Martin Foltin, Suren Byna, and Paolo Faraboschi. Data-Aware Storage Tiering for Deep Learning. In *IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pages 23–28, 2021. doi: 10.1109/PDSW54622.2021.00009.

[6] Sunwoo Lee, Qiao Kang, Kewei Wang, Jan Balewski, Alex Sim, Ankit Agrawal, Alok Choudhary, Peter Nugent, Kesheng Wu, and Wei-keng Liao. Asynchronous I/O Strategy for Large-Scale Deep Learning Applications. In *IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 322–331, 2021. doi: 10.1109/HiPC53243.2021.00046.

[7] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Andrew E. Shao, Scott Bachman, Gustavo Marques, and Benjamin Robbins. Using Machine Learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling. *Journal of Computational Science*, 62:101707, 2022. ISSN 1877-7503. doi: 10.1016/j.jocs.2022.101707.

[8] David Reinsel, John Gantz, and John Rydning. The Digitization of the World From Edge to Core. Technical report, International Data Corporation (IDC), 5 Speen Street Framingham, MA 01701 USA, 2018.

[9] Petroc Taylor. Amount of data created, consumed, and stored 2010-2023, with forecasts to 2028. `https://bit.ly/4fjawMz`, November 2024. Accessed: 2024-12-01.

[10] Yusuke Tanimura, Rosa Filgueira, Isao Kojima, and Malcolm Atkinson. Two-Phase Collective I/O based on Advanced Reservations to Obtain Performance Guarantees from Shared Storage Systems. In *Proceedings of the 2013 Workshop on Interfaces and Abstractions for Scientific Data Storage*, IASDS '13, Indianapolis, Indiana, USA, 2013. IEEE Computer Society.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI 12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[12] Haoyuan Li, Ali Ghodsi, Scott Shenker, Ion Stoica, and Matei A. Zaharia. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, New

York, NY, USA, 2014. Association for Computing Machinery (ACM). ISBN 9781450332521. doi: 10.1145/2670979.2670985.

[13] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Communication ACM*, 54(7):121–130, July 2011. ISSN 0001-0782.

[14] Peng Cheng, Yutong Lu, Yunfei Du, Zhiguang Chen, and Yang Liu. Optimizing Data Placement on Hierarchical Storage Architecture via Machine Learning. In Xiaoxin Tang, Quan Chen, Pradip Bose, Weiming Zheng, and Jean-Luc Gaudiot, editors, *16th IFIP International Conference on Network and Parallel Computing (NPC)*, volume LNCS-11783 of *Network and Parallel Computing*, pages 289–302, Hohhot, China, August 2019. Springer International Publishing. doi: 10.1007/978-3-030-30709-7_23.

[15] Jiang Zhou, Yong Chen, Mai Zheng, and Weiping Wang. Data Distribution for Heterogeneous Storage Systems. *IEEE Transactions on Computers*, 72(6):1747–1762, 2023. doi: 10.1109/TC.2022.3223302.

[16] Dan Huang, Zhenlu Qin, Qing Liu, Norbert Podhorszki, and Scott Klasky. Identifying challenges and opportunities of in-memory computing on large HPC systems. *Journal of Parallel and Distributed Computing*, 164:106–122, 2022. ISSN 0743-7315. doi: 10.1016/j.jpdc.2022.02.002.

[17] AMPLab. Berkeley Data Analytics Stack. `https://amplab.cs.berkeley.edu/software/`, 2024. Accessed: 2024-03-08.

[18] The Apache Software Foundation. Apache Spark Documentation. `https://spark.apache.org/`, 2024. Accessed: 2024-03-08.

[19] The Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`, 2024. Accessed: 2024-03-08.

[20] Margaret J. Wright. Report on Exascale Computing, The ASCAC Subcommittee on Exascale Computing. https://bit.ly/2P1s9cE, September 2010. Accessed: 2024-04-29.

[21] Oracle. Oracle TimesTen In-Memory Database Architectural Overview. https://bit.ly/3UERdGv, March 2019. Accessed: 2024-04-29.

[22] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Engineering Bulletin*, 36:14–20, 2013.

[23] Volt Active Data, Inc. Volt Active Data. https://docs.voltactivedata.com/, 2023. Accessed: 2024-04-30.

[24] SAP HANA. What is SAP HANA? https://bit.ly/3y0ez0b, September 2020. Accessed: 2024-04-29.

[25] P. Chris Broekema, Rob V. van Nieuwpoort, and Henri E. Bal. ExaScale high performance computing in the square kilometer array. In *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, Astro-HPC '12, pages 9–16, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 978-1-4503-1338-4.

[26] SKA Project. The Square Kilometre Array Project. http://www.skatelescope.org/project/, November 2024. Accessed: 2024-11-20.

[27] David Calvet. Upgrade of ATLAS hadronic tile calorimeter for the high luminosity LHC. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 1066:169614, 2024. ISSN 0168-9002. doi: 10.1016/j.nima.2024.169614.

[28] Gottbrath Christopher, Barrett Brian, D. Gropp William, Ewing L. Lusk, and Jeffrey M. Squyres. An Interface to Support the Identification of Dynamic MPI 2 Processes for Scalable Parallel Debugging. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, pages 115–122, Bonn, Germany, September 2006.

[29] MPICH.ORG. MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. http://www.mpich.org/, 2016. Accessed: 2024-03-20.

[30] LBNL and UC Berkeley. Berkeley UPC - Unified Parallel C. https://upc.lbl.gov/, March 2024. Accessed: 2024-04-30.

[31] Yifeng Luo, Siqiang Luo, Jihong Guan, and Shuigeng Zhou. A RAMCloud Storage System based on HDFS: Architecture, implementation and evaluation. *Journal of Systems and Software*, 86(3):744–750, March 2013. ISSN 0164-1212.

[32] Rutgers Discovery Informatics Institute. DataSpaces: An Extreme Scale Data Management Framework. https://dataspaces.sci.utah.edu/, 2024. Accessed: 2024-04-25.

[33] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589428. doi: 10.1145/1851476.1851481.

[34] NVIDIA. RDMA over Converged Ethernet (RoCE). https://bit.ly/3w3xoiF, February 2024. Accessed: 2024-04-29.

[35] Intel Corporation. iWARP RDMA Here and Now. https://intel.ly/4gm53Ga, May 2018. Accessed: 2024-06-18.

[36] Azam Fazel-Najafabadi, Mahdi Abbasi, Hani H. Attar, Ayman Amer, Amir Taherkordi, Azad Shokrollahi, Mohammad R. Khosravi, and Ahmed A. Soyman. High-Performance Flow Classification of Big Data Using Hybrid CPU-GPU Clusters of Cloud Environments. *Tsinghua Science and Technology*, 29(4):1118–1137, 2024. doi: 10.26599/TST.2023.9010088.

[37] Julian Morillo, Maxime Vassaux, Peter V. Coveney, and Marta Garcia-Gasulla. Hybrid parallelization of molecular dynamics simulations to reduce load imbalance. *The Journal of Supercomputing*, 78(7):9184–9215, 2022. doi: 10.1007/s11227-021-04214-4.

[38] Rochan Avlur Venkat, Zakaria Oussalem, and Arya Kumar Bhattacharya. Training Convolutional Neural Networks with Differential Evolution using Concurrent Task Apportioning on Hybrid CPU-GPU Architectures. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2567–2576, 2021. doi: 10.1109/CEC45853.2021.9504878.

[39] Qinnan Qiu, Yongmei Lei, Dongxia Wang, and Guozheng Wang. An efficient hybrid MPI/OpenMP parallelization of the asynchronous ADMM algorithm. In *IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 563–570, 2021. doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00083.

[40] Niclas Jansson. A Hybrid MPI+PGAS Approach to Improve Strong Scalability Limits of Finite Element Solvers. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 303–313, 2020. doi: 10.1109/CLUSTER49012.2020.00041.

[41] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with MPI and unified parallel C. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 177–186, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0044-5.

[42] Vinod Tipparaju, William Gropp, Hubert Ritzdorf, Rajeev Thakur, and Jesper L. Traff. Investigating High Performance RMA Interfaces for the MPI-3 Standard. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 293–300, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0.

[43] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal High Performance Computing Networking*, 1(1-3):91–99, August 2004. ISSN 1740-0562.

[44] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified

parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[45] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. UPC++: A high-performance communication framework for asynchronous computation. *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, pages 963–973, 2019. doi: 10.1109/IPDPS.2019.00104.

[46] Guy L. Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. *Fortress (Sun HPCS Language)*, pages 718–735. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_190.

[47] D. Callahan, B.L. Chamberlain, and H.P. Zima. The Cascade High Productivity Language - Chapel. In *Proceedings of Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004. doi: 10.1109/HIPS.2004.1299190.

[48] Olivier Tardieu. X10 at scale. In *Proceedings of the Third ACM SIGPLAN X10 Workshop*, X10 '13, page 30, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321570. doi: 10.1145/2481268.2481276.

[49] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. ISSN 1061-7264. doi: 10.1145/289918.289920.

[50] Pacific Northwest National Laboratory. Global Arrays Toolkit. `http://hpc.pnl.gov/globalarrays/`, April 2017. Accessed: 2024-04-28.

[51] Dan Bonachea. GASNet specification v1.8. Technical report, LBNL FTG and U.C. Berkeley, August 2017.

[52] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC:Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005. ISBN 3 978-0-471-22048-0.

[53] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Transactions on Parallel Computing*, 5(4):15:1–15:27, March 2019. ISSN 2329-4949. doi: 10.1145/3309205.

[54] Philip H. Carns, Walter B. III Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.

[55] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, W. Liao, A. Choudhary, M. Parashar, C. Docan, and R. Oldfield. Adaptive IO System (ADIOS). In *Cray User's Group*, 2008.

[56] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581569. doi: 10.1145/1383529.1383533.

[57] Jay Lofstead, Karsten Schwan, Fang Zheng, and Scott Klasky. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2009.

[58] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2010. ACM.

[59] Yusuke Tanimura, Rosa Filgueira, Isao Kojima, and Malcolm Atkinson. Abstract: Reservation-Based I/O Performance Guarantee for MPI-IO Applications Using Shared Storage Systems. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1382–1383, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4956-9.

[60] Fraunhofer. BeegGFS: A Fast and Scalable Parallel Filesystem. `https://www.beegfs.io/c/`, March 2024. Accessed: 2024-04-29.

[61] The Gluster Community. GlusterFS. http://www.gluster.org/, 2019. Accessed: 2024-04-15.

[62] Sun Microsystems, Inc., Santa Clara, CA, USA. Lustre file system - High-performance storage architecture and scalable cluster file system. https://bit.ly/3UBTXEp, 2007. Accessed: 2024-04-15.

[63] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3), August 2015. ISSN 0734-2071. doi: 10.1145/2806887.

[64] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 2nd edition, 1999. ISBN 026257134X.

[65] Prabhat and Quincey Koziol. *High Performance Parallel I/O*. Chapman & Hall/CRC, 1st edition, 2014. ISBN 1466582340.

[66] R Thakur, E Lusk, and W Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical report, Argonne National Laboratory, May 2004.

[67] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 180–187, 1996.

[68] Tarek El-Ghazawi, François Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, and Dan Bonachea. UPC-IO: A Parallel I/O API for UPC V1.0. Technical report, The George Washington University, 2004.

[69] Norbert Podhorszki, Qing Liu, Jeremy Logan, Jingqing Mu, Hasan Abbasi, Jong-Youl Choi, and Scott A. Klask. ADIOS 1.11 User's Manual. Technical report, Oak Ridge National Laboratory & Georgia Institute of Technology, 2016.

[70] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck,

Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX*, 12:100561, 2020. ISSN 2352-7110. doi: 10.1016/j.softx.2020.100561.

[71] Haoyuan Li. *Alluxio: A Virtual Distributed File System*. PhD thesis, EECS Department, University of California, Berkeley, May 2018.

[72] Memcached Community. Memcached. `http://memcached.org/`, November 2018. Accessed: 2024-04-15.

[73] UChicago Argonne, LLC and The HDF Group. Mercury. `https://mercury-hpc.github.io/`, April 2022. Accessed: 2024-05-10.

[74] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Trans. Storage*, 12(4), May 2016. ISSN 1553-3077. doi: 10.1145/2846100.

[75] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient Memory Mapped File I/O for In-Memory File Systems. In *Proceedings - 9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. {USENIX} Association.

[76] Jian Xu and Steven Swanson. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, USA, 2016. USENIX Association. ISBN 9781931971287.

[77] Apache Software Foundation. In-Memory Database With Apache Ignite. `https://bit.ly/3yX28hi`, April 2024. Accessed: 2024-04-30.

[78] Robert Ross, Lee Ward, Philip Carns, Gary Grider, Scott Klasky, Quincey Koziol, Glenn K. Lockwood, Kathryn Mohror, Bradley W. Settlemyer, and Matthew Wolf. Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific

Discovery. Technical report, Advanced Computing Research, USA, Gaithersburg, Maryland, 2019.

[79] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick Mccormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, et al. *S3D-Legion: An Exascale Software for Direct Numerical Simulation of Turbulent Combustion with Complex Multicomponent Chemistry*. CRC Press, Taylor & Francis Group, Boca Raton, Florida,United States of America, November 2017. doi: 10.1201/b21930-12.

[80] J. T. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, February 2006. ISSN 0167-739X.

[81] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, USA, 2010. IEEE Computer Society. ISBN 9781424475599. doi: 10.1109/SC.2010.18.

[82] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance Fault Tolerance Interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–12, 2011. doi: 10.1145/2063384.2063427.

[83] Berkely Lab. Berkeley Lab Checkpoint/Restart (BLCR) User's Guide. `https://bit.ly/4fsFDFf`, May 2023. Accessed: 2024-04-29.

[84] Lawrence Livermore National Laboratory. PCMDI - Program for Climate Model Diagnosis & Intercomparison. `https://pcmdi.llnl.gov/`, February 2022. Accessed: 2024-04-30.

[85] Rene Brun and Fons Rademakers. ROOT — An object-oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997. ISSN

0168-9002. doi: 10.1016/S0168-9002(97)00048-X. New Computing Techniques in Physics Research V.

[86] Benjamin R. Landsteiner, Dave Henseler, Doug Petesch, and N. Wright. Architecture and Design of Cray DataWarp. In *2016 Cray User Group*, London, 2016.

[87] Terabit Systems. InfiniBand vs Ethernet: A Technical Comparison for HPC and AI Compute Connectivity. https://bit.ly/3ZQwcuM, Oct 2024. Accessed: 2024-11-12.

[88] Mellanox Technologies. Introducing 200G HDR InfiniBand Solutions White Paper. https://bit.ly/4bh6Ucc, March 2019. Accessed: 2024-04-30.

[89] Intel, Inc. Intel Omni-Path Fabric Suite. https://intel.ly/3UGuMAz, March 2020. Accessed: 2024-04-30.

[90] Milan Radulovic, Kazi Asifuzzaman, Paul Carpenter, Petar Radojković, and Eduard Ayguadé. HPC Benchmarking: Scaling Right and Looking Beyond the Average. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 135–146, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96983-1.

[91] Mellanox Technologies. RoCE vs. iWARP Competitive Analysis. https://bit.ly/3qPzry1, March 2017. Accessed: 2024-03-15.

[92] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and Atomic RDMA-Based Replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 851–863, USA, 2018. USENIX Association. ISBN 9781931971447.

[93] NASA Advanced Supercomputing. The NAS parallel benchmarks. https://go.nasa.gov/3JJDZlp, July 2023. Accessed: 2023-07-01.

[94] Jjnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021. ISSN 0167-739X. doi: 10.1016/j.future.2021.07.021.

[95] Tarek El-Ghazawi and Francois Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–26, Washington, DC, USA, 2002. IEEE Computer Society Press. ISBN 076951524X.

[96] François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed S. Mohamed, and Tarek A. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, page 274.2, USA, 2003. IEEE Computer Society. ISBN 0769519261. doi: 10.1109/IPDPS.2003.1213492.

[97] Zhang Zhang and Steven Seidel. Benchmark Measurements of Current UPC Platforms. In *Proceedings of 19th International Parallel and Distributed Processing Symposium*, January 2005. doi: 10.1109/IPDPS.2005.123.

[98] Manojkumar Krishnan and Jarek Nieplocha. SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 70–, April 2004. doi: 10.1109/IPDPS.2004.1303000.

[99] Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[100] Jaeyoung Choi, David W. Walker, and Jack J. Dongarra. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6:543–570, 1994. doi: 10.1002/cpe.4330060702.

[101] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. Generalized Cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 44–51, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919025. doi: 10.1145/263580.263591.

[102] Hsin-Chen Lu, Liang-Ying Su, and Shih-Hsu Huang. Highly Fault-Tolerant Systolic-Array-Based Matrix Multiplication. *Electronics*, 13(9), 2024. ISSN 2079-9292. doi: 10.3390/electronics13091780.

[103] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New Bounds for Matrix Multiplication: from Alpha to Omega, 2023.

[104] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022. doi: 10.1038/s41586-022-05172-4.

[105] Hua Huang and Edmond Chow. CA3DMM: A New Algorithm Based on a Unified View of Parallel Matrix Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '22, pages 1–15, 2022. doi: 10.1109/SC41404.2022.00033.

[106] Weiqi Li, Zhen Chen, Zhiying Wang, Syed A. Jafar, and Hamid Jafarkhani. Flexible Distributed Matrix Multiplication. *IEEE Transactions on Information Theory*, 68 (11):7500–7514, 2022. doi: 10.1109/TIT.2022.3204488.

[107] Edgar Solomonik and James Demmel. Matrix Multiplication on Multidimensional Torus Networks. In *High Performance Computing for Computational Science - VECPAR 2012*, pages 201–215, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38718-0.

[108] Keqin Li. Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers. *Journal of Parallel and Distributed Computing*, 61(12):1709–1731, 2001. ISSN 0743-7315. doi: 10.1006/jpdc.2001.1768.

[109] Keqin Li. Fast and highly scalable parallel computations for fundamental matrix problems on distributed memory systems. *The Journal of Supercomputing*, 54(3): 271–297, 2010. doi: 10.1007/s11227-009-0319-0.

[110] Luisa DÁmore, Giuliano Laccetti, and Marco Lapegna. Block Matrix Multiplication in a Distributed Computing Environment: Experiments with Netsolve. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM05, pages 625–632, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34141-2, 978-3-540-34141-3. doi: 10.1007/11752578_75.

[111] Anshul Gupta and Vipin Kumar. Scalability of Parallel Algorithms for Matrix Multiplication. In *International Conference on Parallel Processing - ICPP'93*, volume 3, pages 115–123, 1993. doi: 10.1109/ICPP.1993.160.

[112] G. Nimako, E. J. Otoo, and D. Ohene-Kwofie. Fast Parallel Algorithms for Blocked Dense Matrix Multiplication on Shared Memory Architectures. In *Algorithms and Architectures for Parallel Processing*, pages 443–457, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33078-0.

[113] University of Tennessee and University of California, Berkeley and University of Colorado, Denver and NAG Ltd. ScaLAPACK - Scalable Linear Algebra PACKage. https://www.netlib.org/scalapack/, February 2022. Accessed: 2024-05-10.

[114] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN 0-89871-397-8.

[115] University of Tennessee and University of California, Berkeley and University of Colorado, Denver and NAG Ltd. LAPACK - Linear Algebra PACKage. https://www.netlib.org/lapack/, May 2024. Accessed: 2024-06-15.

[116] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0.

[117] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, pages 224–229, April 1997. doi: 10.1109/HPC.1997.592151.

[118] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8.

[119] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency: Practice and Experience*, 22(1):15–44, January 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:1.

[120] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13 (4):354–356, 1969. ISSN 0029-599X. doi: 10.1007/BF02165411.

[121] Paolo D'Alberto and Alexandru Nicolau. Adaptive Strassen's Matrix Multiplication. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 284–292, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-768-1. doi: 10.1145/1274971.1275010.

[122] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990. ISSN 0747-7171. doi: 10.1016/S0747-7171(08)80013-2.

[123] Christopher Barton, Călin Caşcaval, George Almasi, Rahul Garg, José Nelson Amaral, and Montse Farreras. Multidimensional Blocking in UPC. In Vikram Adve, María Jesús Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 47–62, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-85261-2.

[124] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

[125] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[126] Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. Cengage Learning, 4 edition, 2006. ISBN 0030105676; 9780030105678.

[127] Frank Y. Shih. *Image Processing and Mathematical Morphology: Fundamentals and Applications*. CRC Press, 1 edition, 2009. ISBN 1420089439; 9781420089431; 9781420089448; 1420089447.

[128] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[129] I.T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, 2 edition, 2010. ISBN 9781441929990; 1441929991.

[130] Viswajeetsinh Jadeja, A L N Rao, Amit Srivastava, Shekhar Singh, Prateek Chaturvedi, and Garima Bhardwaj. Convolutional Neural Networks: A Comprehensive Review of Architectures and Application. In *International Conference on Contemporary Computing and Informatics (IC3I)*, volume 6, pages 460–467, 2023. doi: 10.1109/IC3I59117.2023.10397695.

[131] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel SAH k-D tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 77–86, Goslar, DEU, 2010. Eurographics Association.

[132] Aritra Chakravorty, William S. Cleveland, and Patrick J. Wolfe. Scalable k-d trees for Distributed data. *CoRR*, abs/2201.08288, 2022.

[133] Amalia Duch, Conrado Martínez, Mercè Pons, and Salvador Roura. Median and Hybrid Median K-Dimensional Trees. In *LATIN 2022: Theoretical Informatics: 15th Latin American Symposium*, pages 38–53, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-20623-8. doi: 10.1007/978-3-031-20624-5_3.

[134] J David MacDonald and Kellogg S Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990. ISSN 1432-2315. doi: 10.1007/BF01911006.

[135] Ingo Wald and Vlastimil Havran. On building fast k-d trees for ray tracing, and on doing that in O(N log N). *IEEE Symposium on Interactive Ray Tracing 2006, Proceedings*, pages 61–69, 2006. doi: 10.1109/RT.2006.280216.

[136] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jialin Liu, Peter Sadowski, Evan Racah, Suren Byna, Craig Tull, Wahid Bhimji, Prabhat, and Pradeep Dubey. PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503. IEEE, May 2016. doi: 10.1109/ipdps.2016.57.

[137] Victor (University of Illinois) Lu. *Multicore Construction Of K-d trees With Applications In Graphics And Vision*. PhD thesis, University of Illinois, Urbana-Champaign, 2013.

[138] Marius Muja and David G. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *International Conference on Computer Vision Theory and Applications*, 2009.

[139] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[140] PCL. The Point Cloud Library. `https://pointclouds.org/`, June 2024. Accessed: 2025-02-20.

[141] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

[142] A. Vedaldi and B. Fulkerson. VLFeat: An Open and Portable Library of Computer Vision Algorithms. `http://www.vlfeat.org/`, 2018. Accessed: 2025-02-20.

[143] Geofabrik GmbH. OpenStreetMap data for this region – South Africa. `https://download.geofabrik.de/africa/south-africa.html`, 2025. Accessed: 2025-09-14.

[144] Osmium. Osmium Tool. `https://osmcode.org/osmium-tool/`, June 2024. Accessed: 2024-07-13.

[145] Mengyao Zhao, Wan Ping Lee, Erik P. Garrison, and Gabor T. Marth. SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. *PLoS ONE*, 8(12), December 2013. ISSN 19326203. doi: 10.1371/JOURNAL.PONE.0082138.